Computer Architecture

Lecture 13: Memory Interference and Quality of Service

Prof. Onur Mutlu

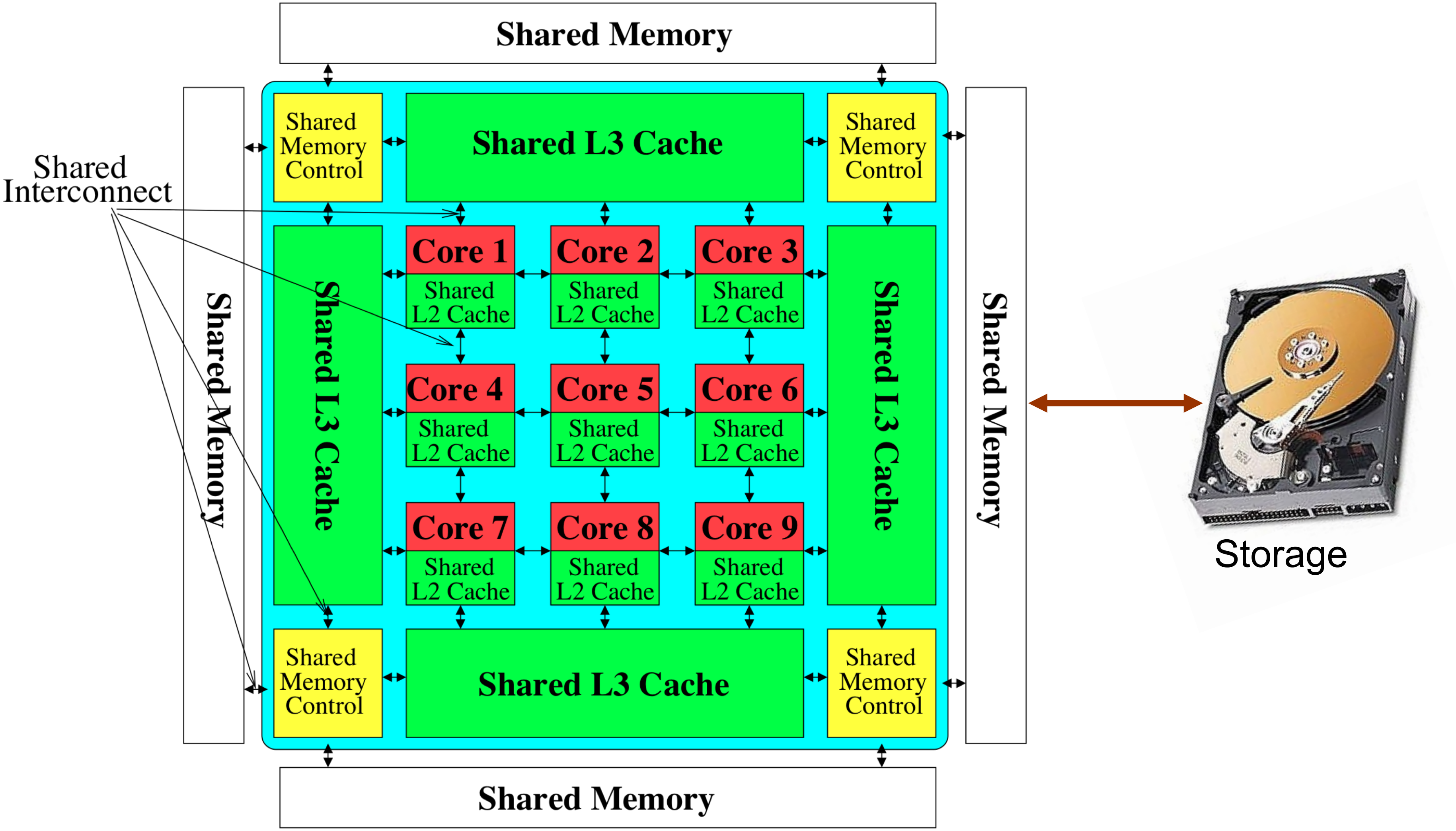
ETH Zürich

Fall 2021

11 November 2021

Shared Resource Design for Multi-Core Systems

Memory System: A ***Shared Resource*** View



Most of the system is dedicated to storing and moving data

Resource Sharing Concept

- Idea: Instead of dedicating a hardware resource to a hardware context, allow multiple contexts to use it
  - Example resources: functional units, pipeline, caches, buses, memory, interconnects, storage
- Why?

+ Resource sharing improves utilization/efficiency → throughput

- When a resource is left idle by one thread, another thread can use it; no need to replicate shared data

+ Reduces communication latency

- For example, shared data kept in the same cache in SMT processors

+ Compatible with the shared memory model

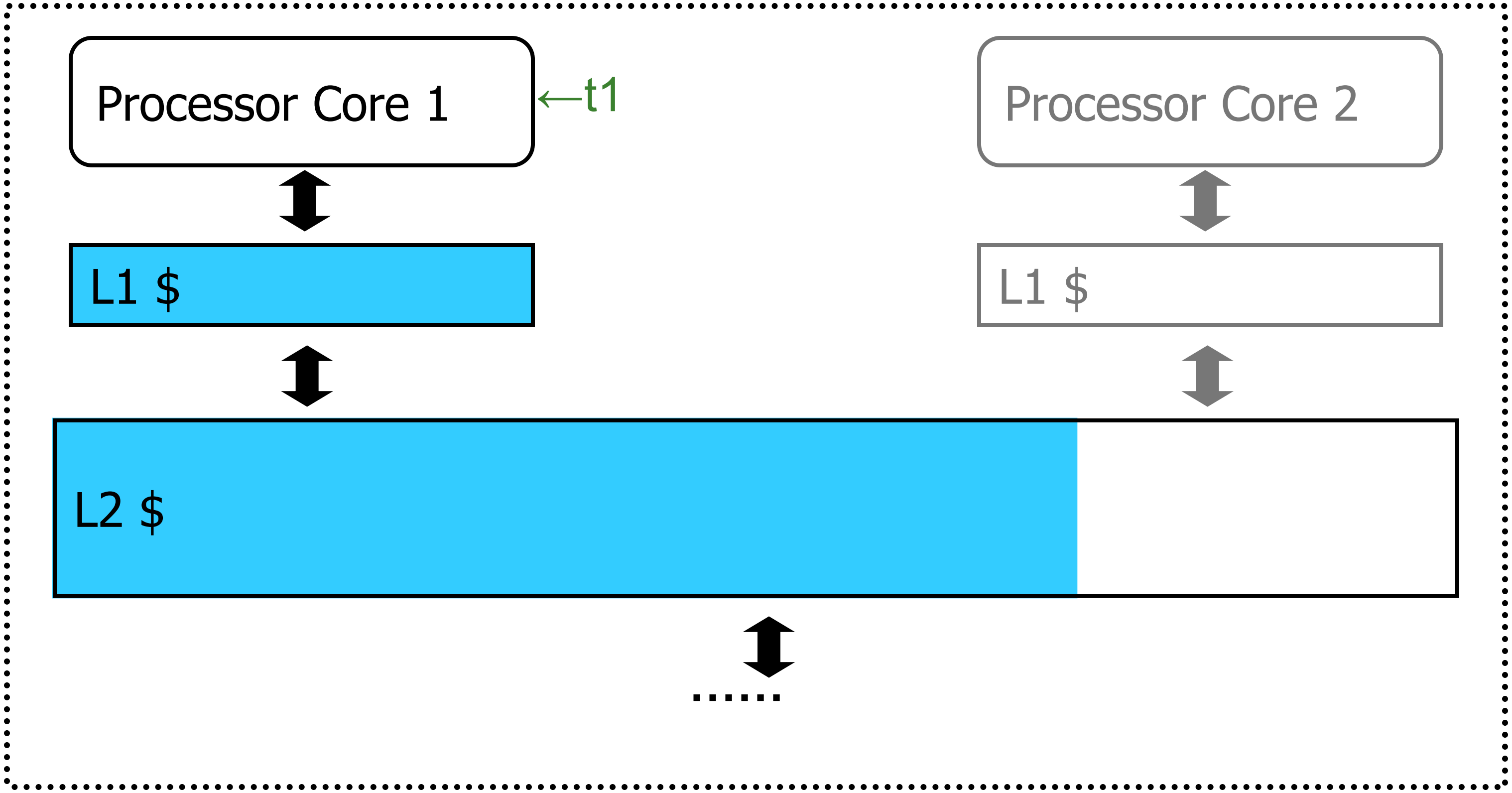
Resource Sharing Disadvantages

- Resource sharing results in contention for resources
  - When the resource is not idle, another thread cannot use it
  - If space is occupied by one thread, another thread needs to re-occupy it

- Sometimes reduces each or some thread's performance
  - Thread performance can be worse than when it is run alone
- Eliminates performance isolation → inconsistent performance across runs
  - Thread performance depends on co-executing threads
- Uncontrolled (free-for-all) sharing degrades QoS
  - Causes unfairness, starvation

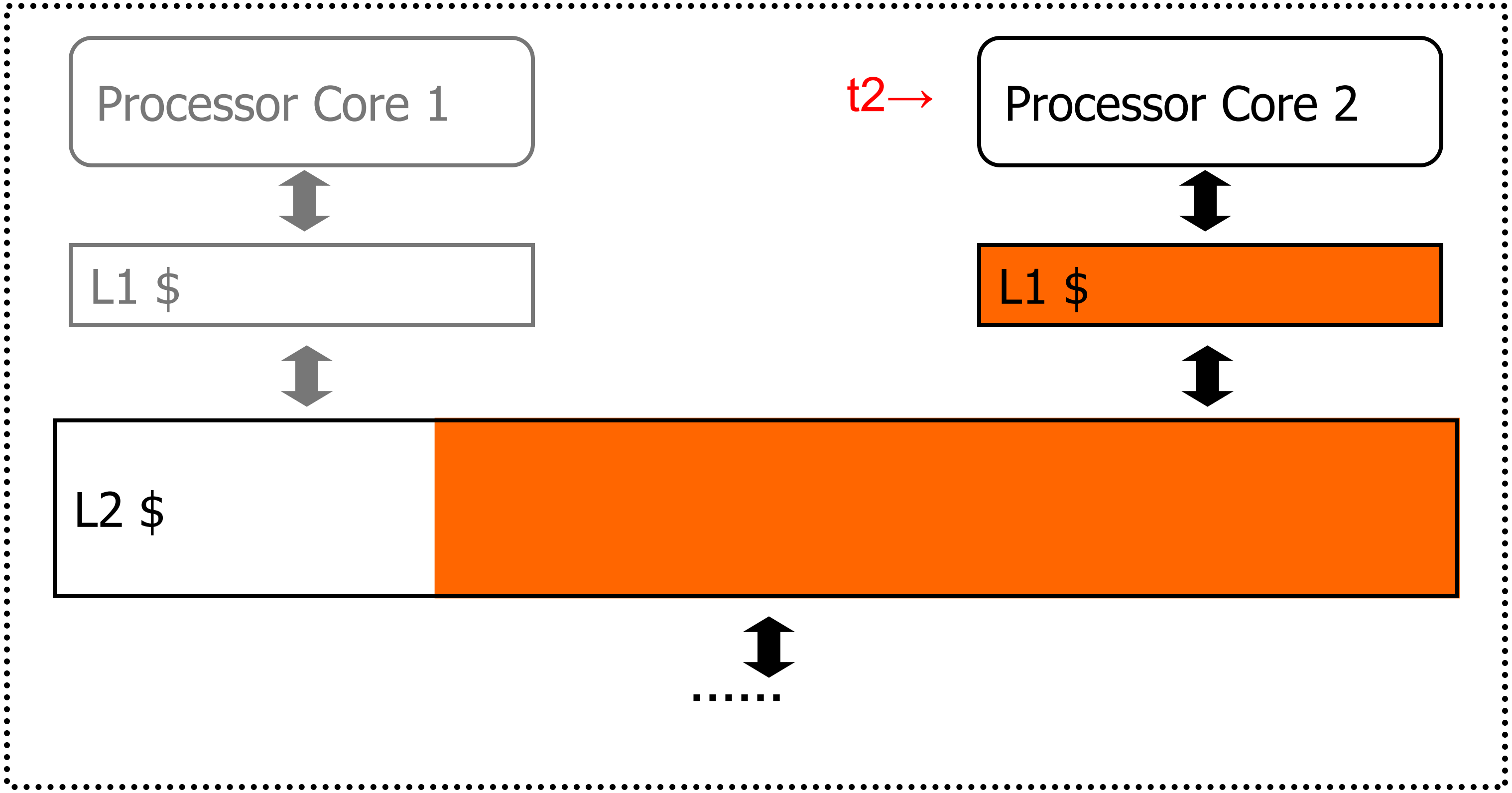
Need to efficiently and fairly utilize shared resources

Example: Problem with Shared Caches



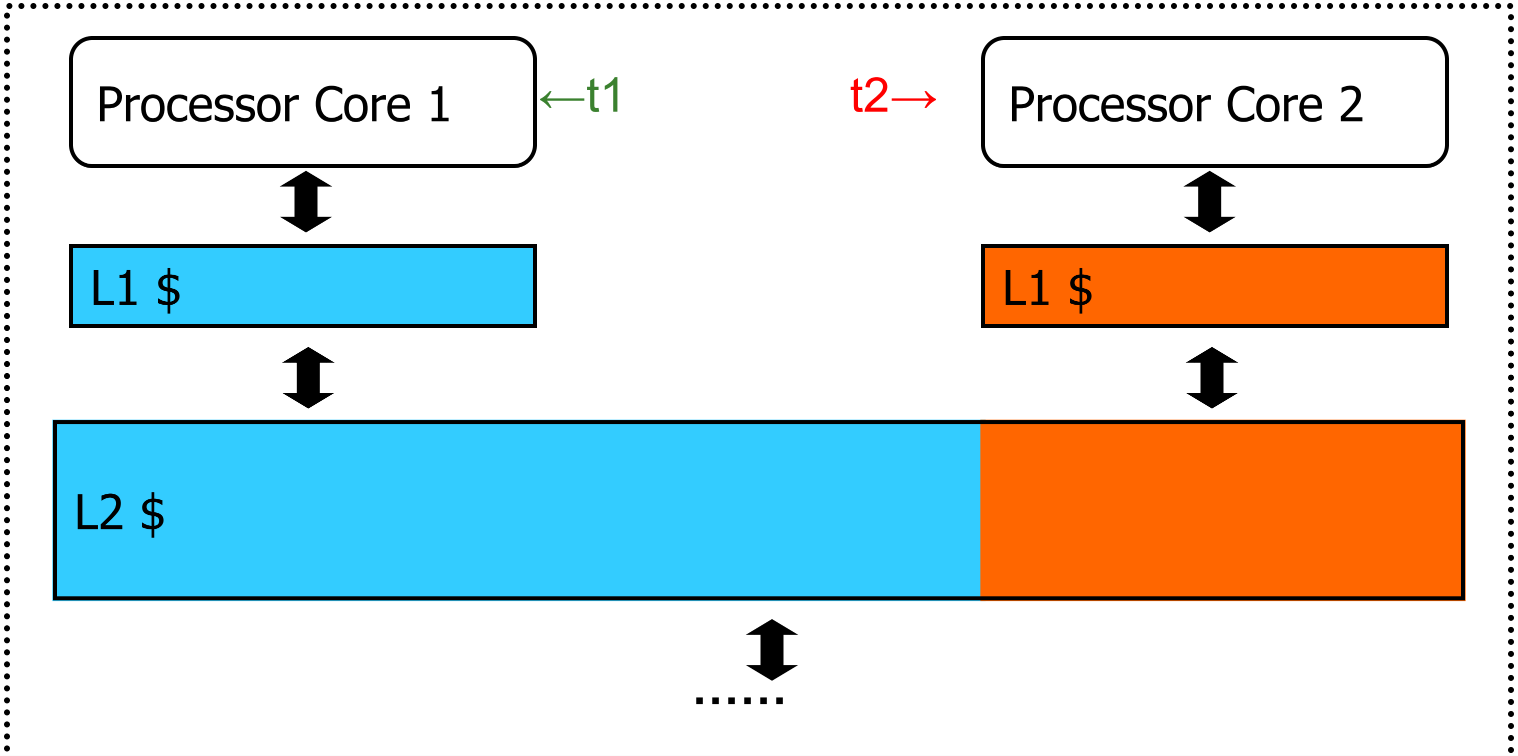
Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: Problem with Shared Caches



Kim et al., "Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture," PACT 2004.

Example: Problem with Shared Caches



t2’s throughput is significantly reduced due to unfair cache sharing.

Kim et al., “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture,” PACT 2004.

Need for QoS and Shared Resource Mgmt.

- Why is unpredictable performance (or lack of QoS) bad?

- Makes programmer's life difficult
  - An optimized program can get low performance (and performance varies widely depending on co-runners)

- Causes discomfort to user
  - An important program can starve
  - Examples from shared software resources

- Makes system management difficult
  - How do we enforce a Service Level Agreement when hardware resources are sharing is uncontrollable?

Resource Sharing vs. Partitioning

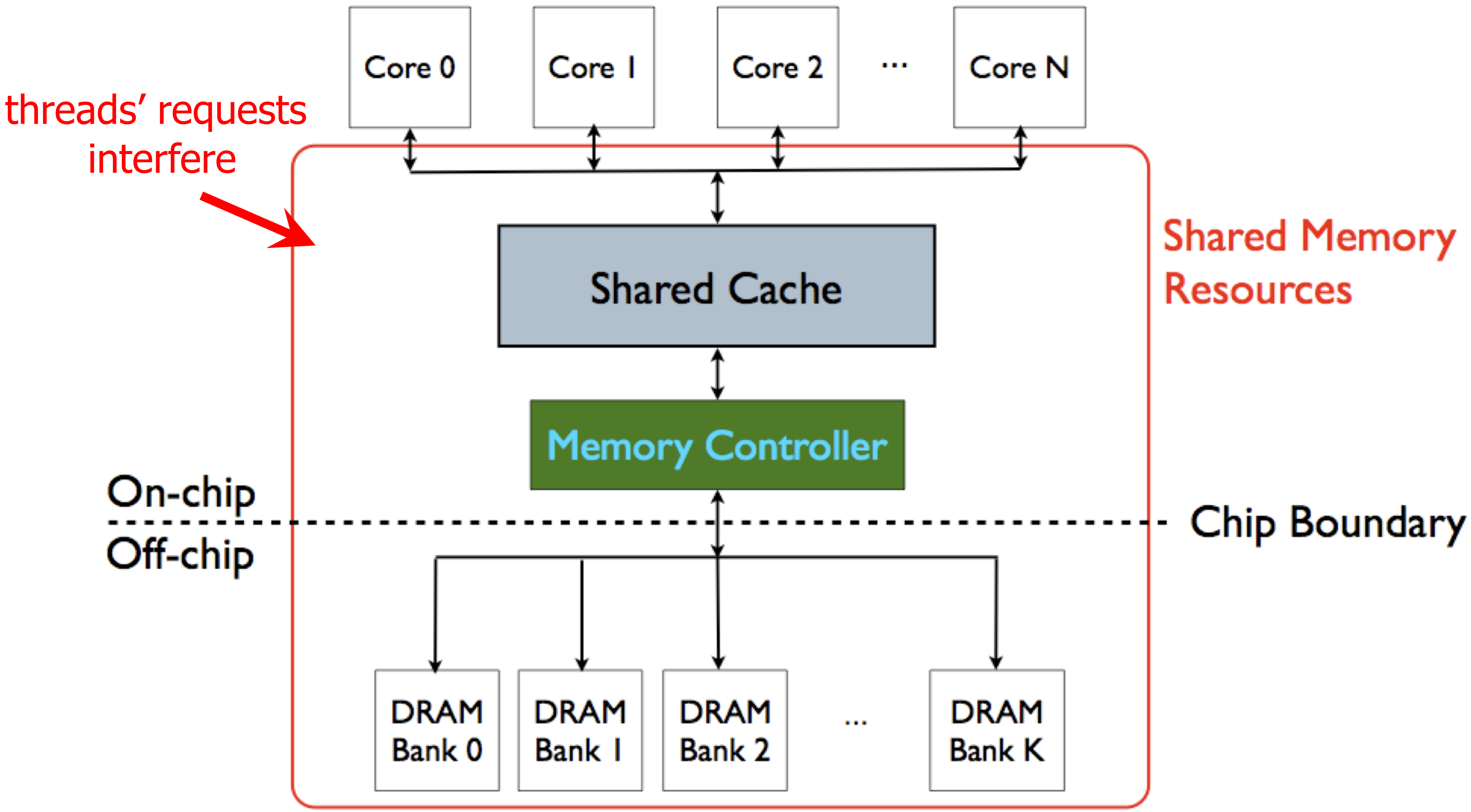
- Sharing improves throughput
  - Better utilization of space

- Partitioning provides performance isolation (predictable performance)
  - Dedicated space

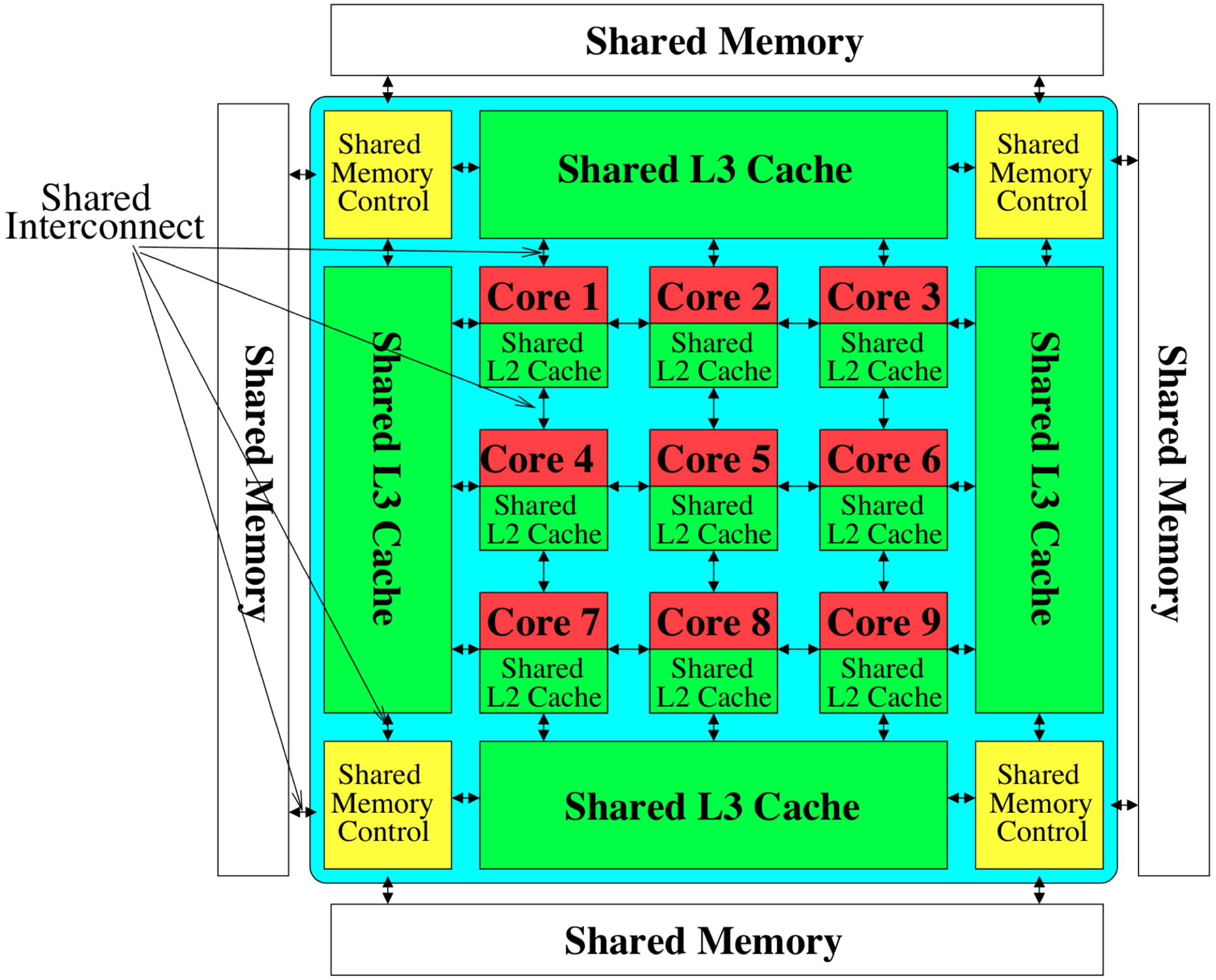
- Can we get the benefits of both?

- Idea: Design shared resources such that they are efficiently utilized, controllable and partitionable
  - No wasted resource + QoS mechanisms for threads

Memory System is the Major Shared Resource



Much More of a Shared Resource in Future

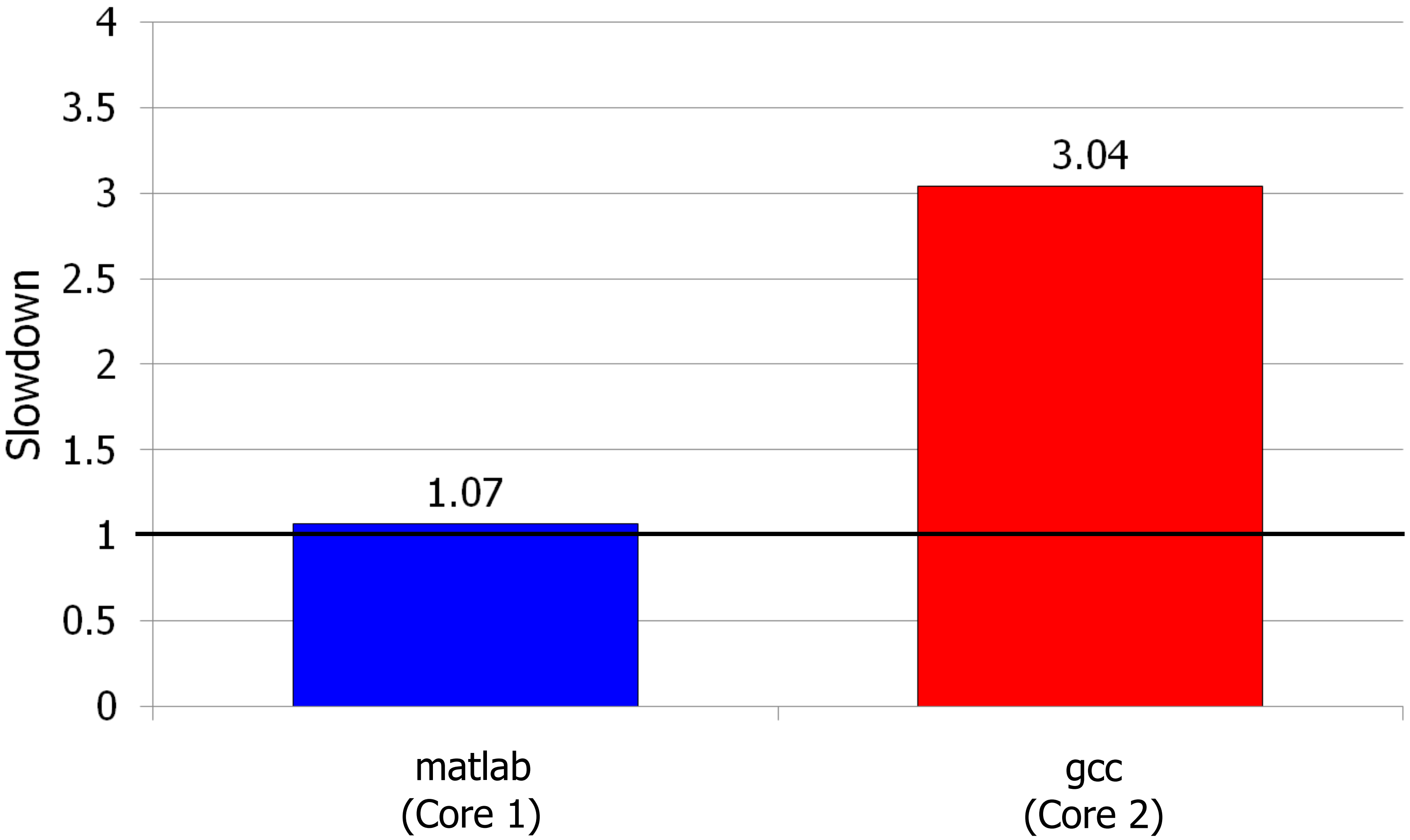


Inter-Thread/Application Interference

- Problem: Threads share the memory system, but memory system does not distinguish between threads' requests

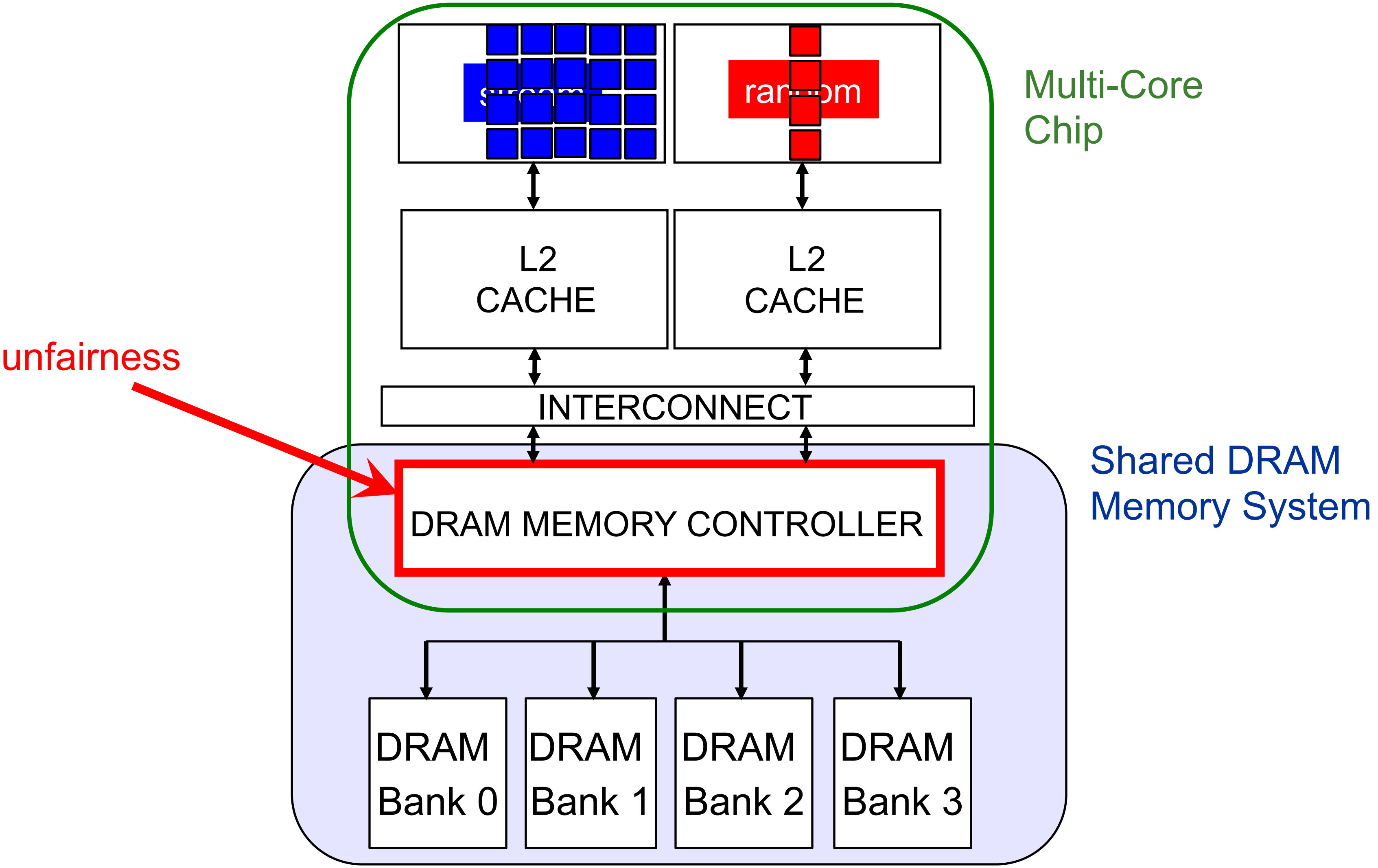
- Existing memory systems
  - Free-for-all, shared based on demand
  - Control algorithms thread-unaware and thread-unfair
  - Aggressive threads can deny service to others
  - Do not try to reduce or control inter-thread interference

Unfair Slowdowns due to Interference



Moscibroda and Mutlu, "Memory performance attacks: Denial of memory service in multi-core systems," USENIX Security 2007.

Uncontrolled Interference: An Example



A Memory Performance Hog

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    …
}
```

streaming

```
// initialize large arrays A, B

for (j=0; j<N; j++) {
    index = rand();
    A[index] = B[index];
    …
}
```

random

STREAM

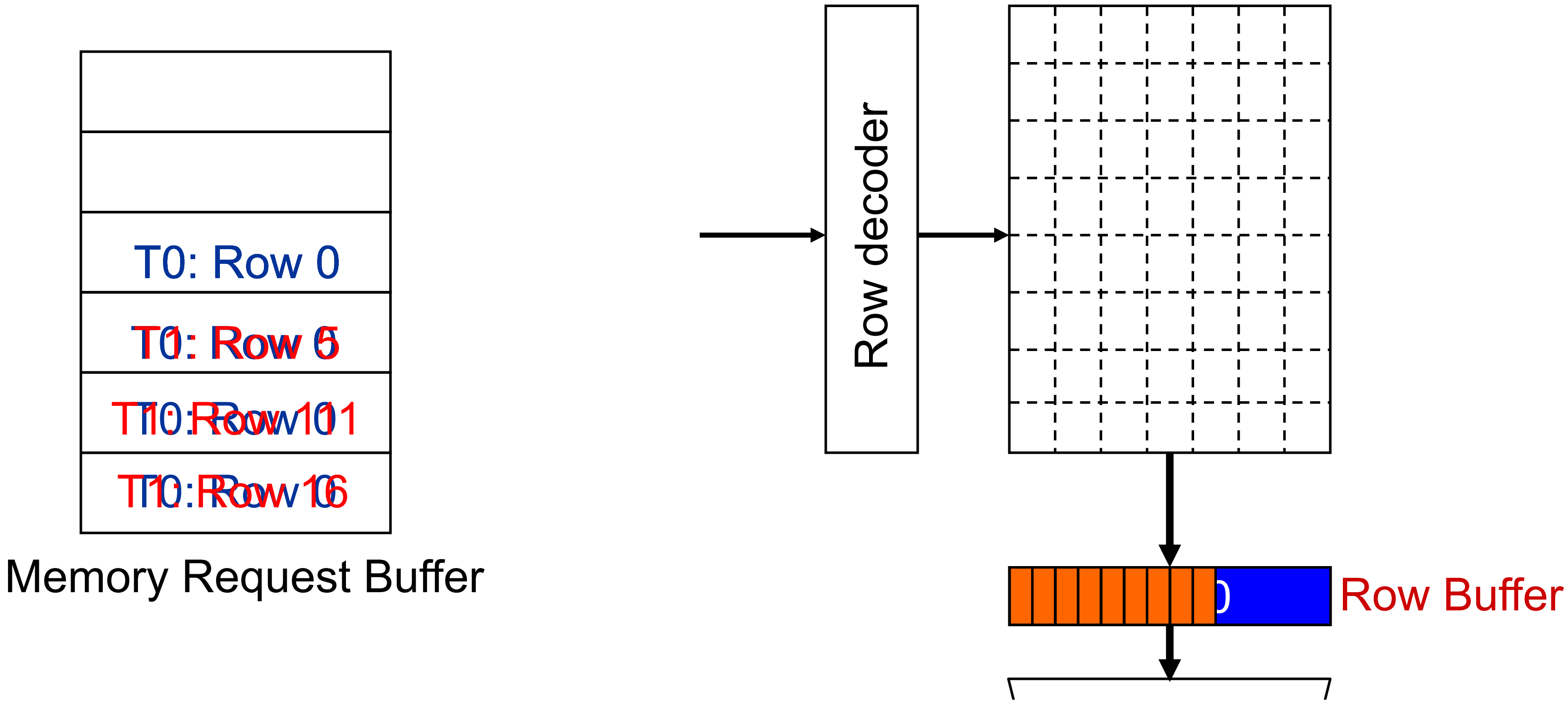
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM

- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

What Does the Memory Hog Do?



Row size: 8KB, cache block size: 64B

128 (8KB/64B) requests of T0 serviced before T1

Moscibroda and Mutlu, “Memory Performance Attacks,” USENIX Security 2007.

DRAM Controllers

- A row-conflict memory access takes significantly longer than a row-hit access

- Current controllers take advantage of the row buffer

- Commonly used scheduling policy (FR-FCFS) [Rixner 2000]*

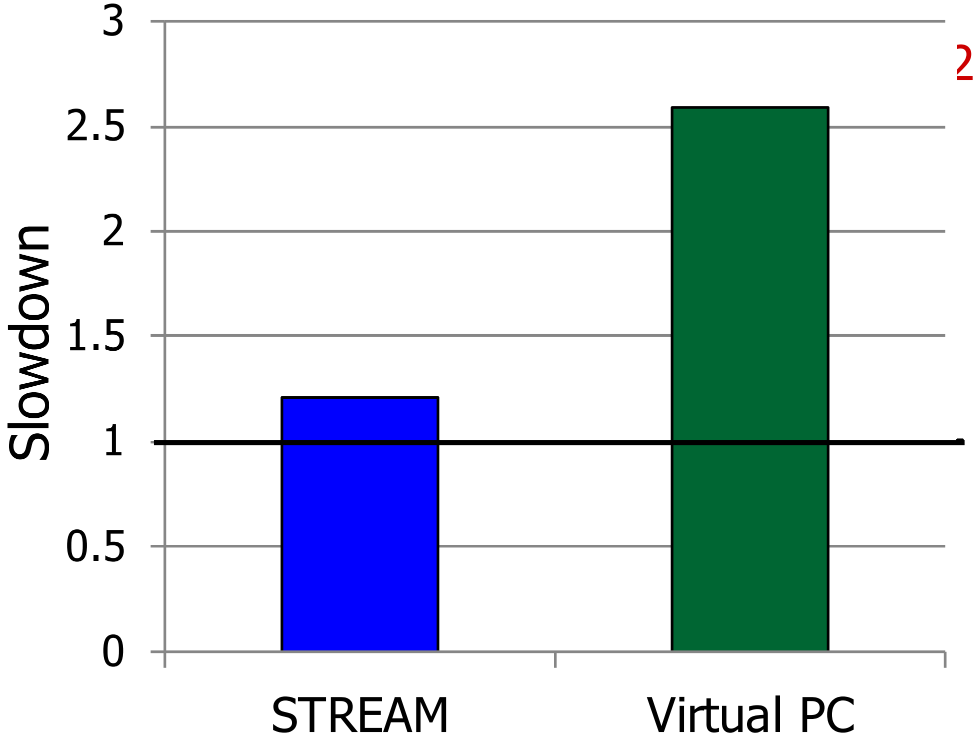
  (1) Row-hit first: Service row-hit memory accesses first

  (2) Oldest-first: Then service older accesses first

- This scheduling policy aims to maximize DRAM throughput
  - But, it is unfair when multiple threads share the DRAM system

*Rixner et al., "Memory Access Scheduling," ISCA 2000.
*Zuravleff and Robinson, "Controller for a synchronous DRAM …," US Patent 5,630,096, May 1997.

Effect of the Memory Performance Hog

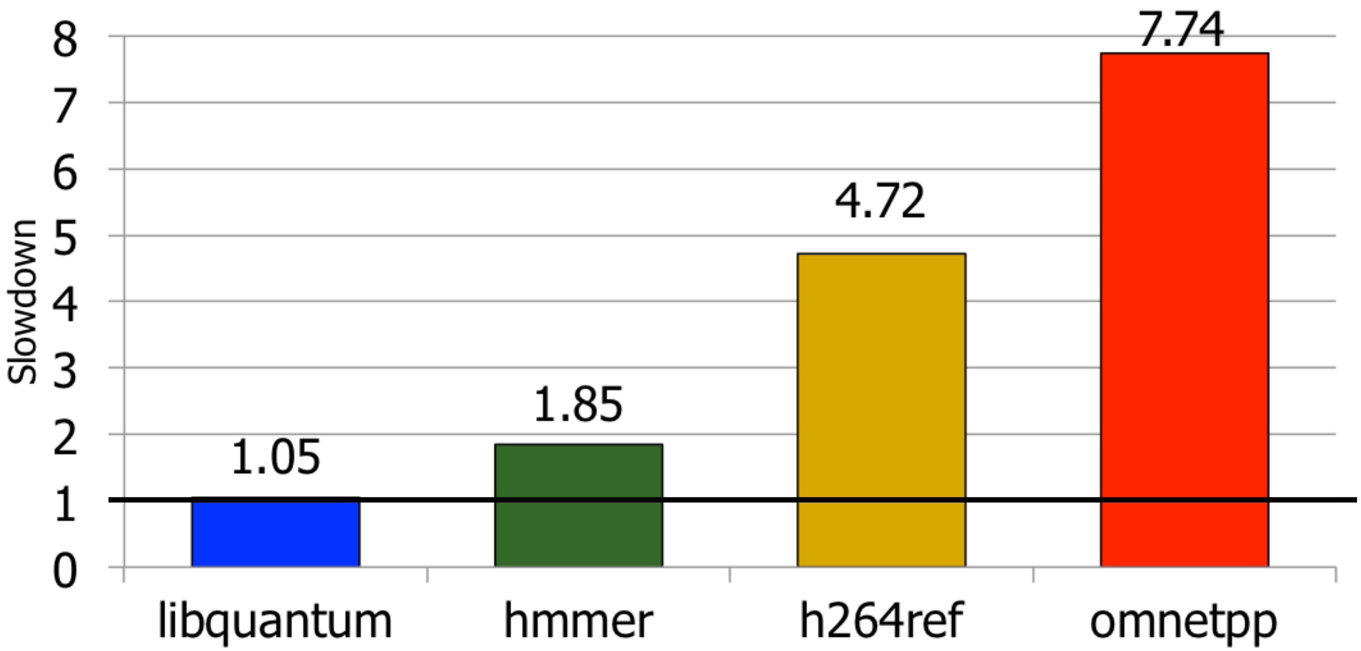


2X slowdown

Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Moscibroda and Mutlu, "Memory Performance Attacks," USENIX Security 2007.

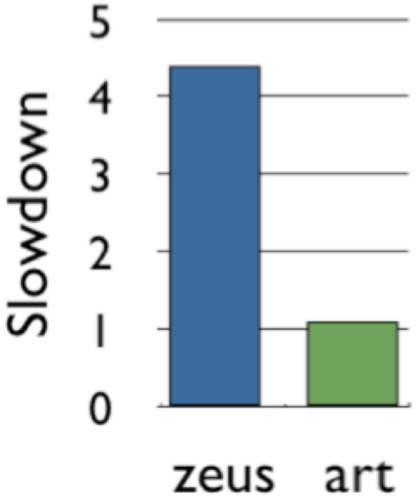
Greater Problem with More Cores

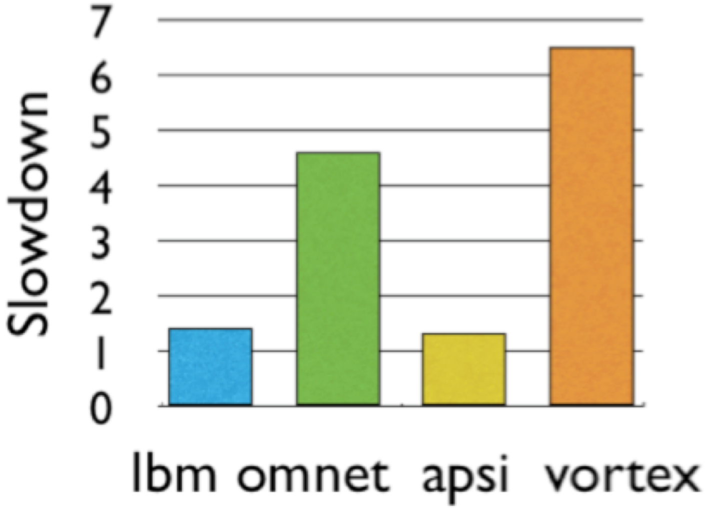


- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system

Greater Problem with More Cores





- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

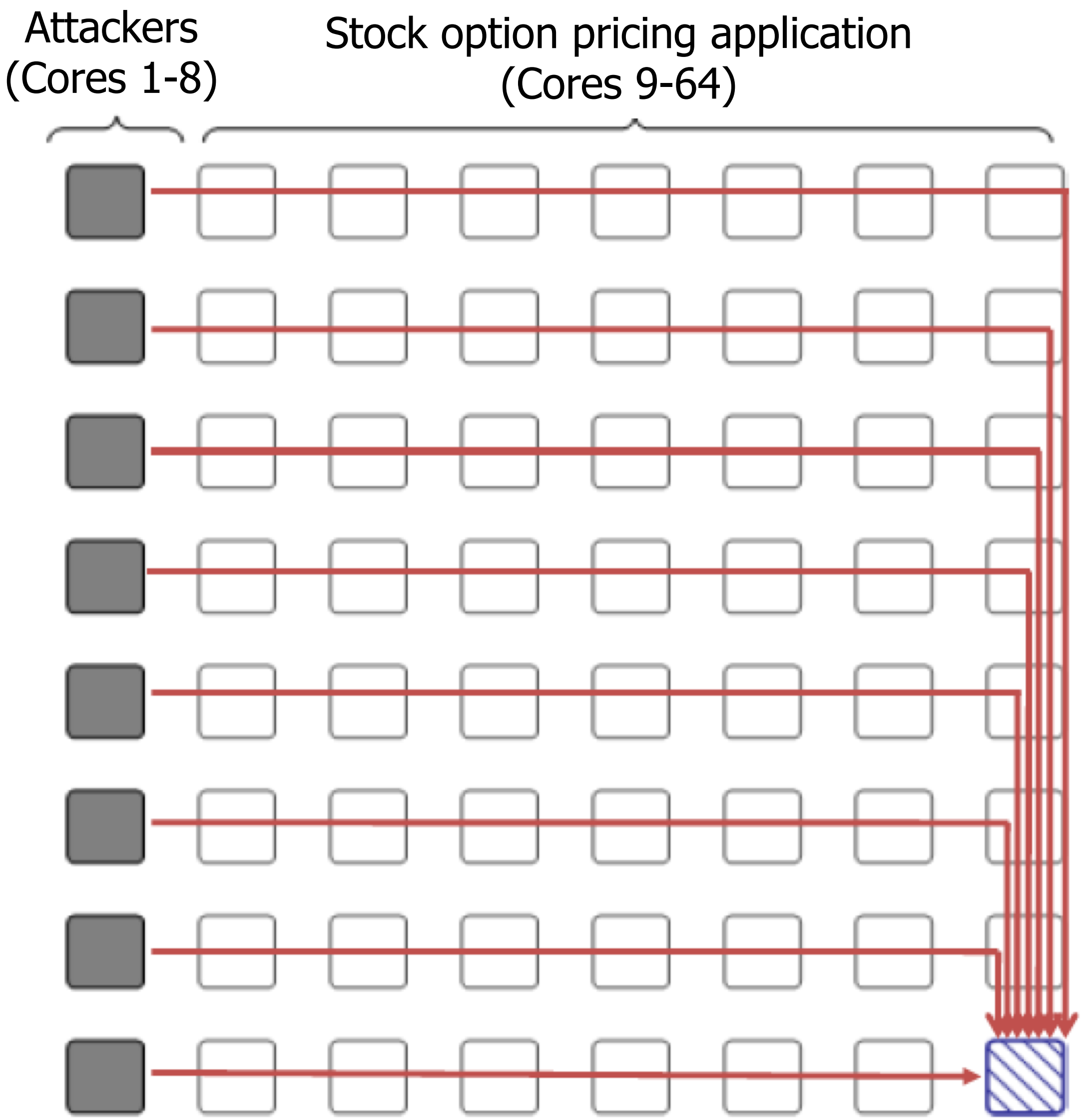
Uncontrollable, unpredictable system

Distributed DoS in Networked Multi-Core Systems

Cores connected via packet-switched routers on chip

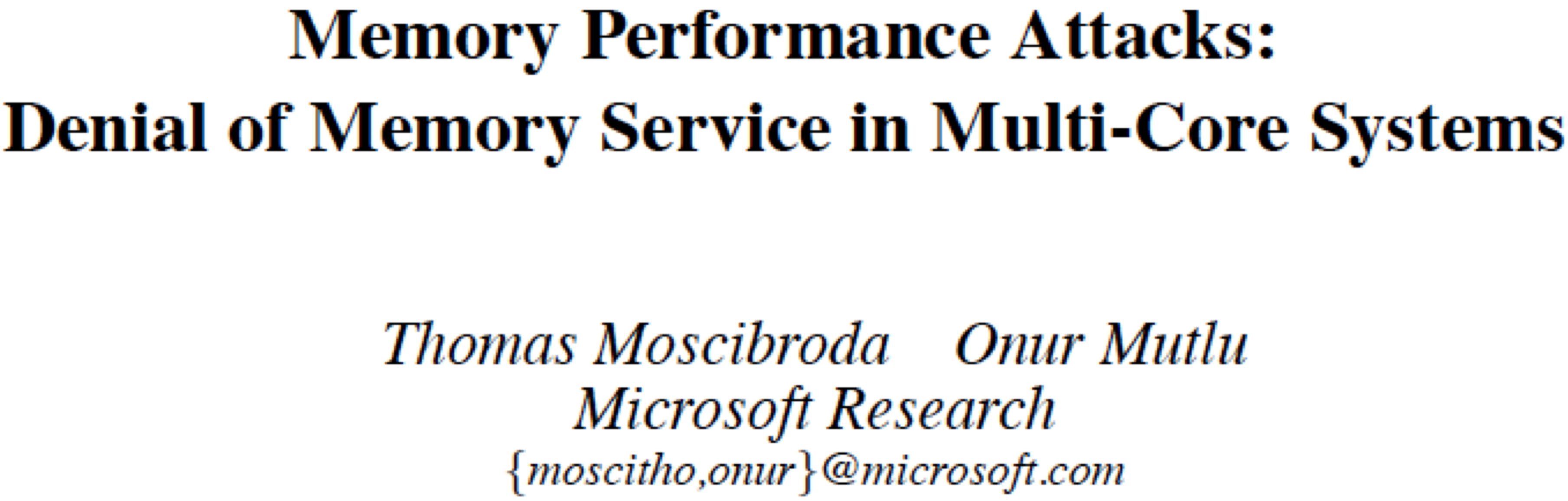
~5000X latency increase

Grot, Hestness, Keckler, Mutlu, "Preemptive virtual clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip," MICRO 2009.



More on Memory Performance Attacks

- Thomas Moscibroda and Onur Mutlu, **"Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems"** *Proceedings of the 16th USENIX Security Symposium* (***USENIX SECURITY***), pages 257-274, Boston, MA, August 2007. Slides (ppt)



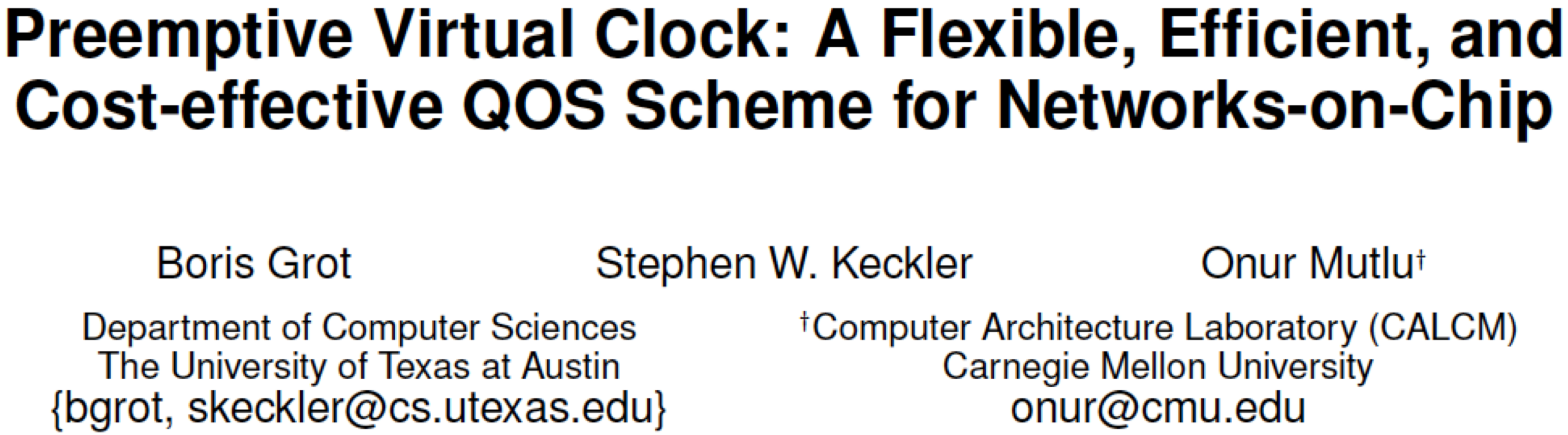
Memory Performance Attacks:
Denial of Memory Service in Multi-Core Systems

Thomas Moscibroda Onur Mutlu
Microsoft Research
{*moscitho,onur*}*@microsoft.com*

http://www.youtube.com/watch?v=VJzZbwgBfy8

More on Interconnect Based Starvation

- Boris Grot, Stephen W. Keckler, and Onur Mutlu, **"Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip"** *Proceedings of the 42nd International Symposium on Microarchitecture* (***MICRO***), pages 268-279, New York, NY, December 2009. Slides (pdf)



Preemptive Virtual Clock: A Flexible, Efficient, and Cost-effective QOS Scheme for Networks-on-Chip

Boris Grot Stephen W. Keckler Onur Mutlu[†]

Department of Computer Sciences
The University of Texas at Austin
{bgrot, skeckler@cs.utexas.edu}

[†]Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches

- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system

QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation
- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance
- How do we make the memory system configurable/flexible?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed

Designing QoS-Aware Memory Systems: Approaches

- Smart resources: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers
  - QoS-aware interconnects
  - QoS-aware caches

- Dumb resources: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system
  - QoS-aware data mapping to memory controllers
  - QoS-aware thread scheduling to cores

Fundamental Interference Control Techniques

- Goal: to reduce/control inter-thread memory interference

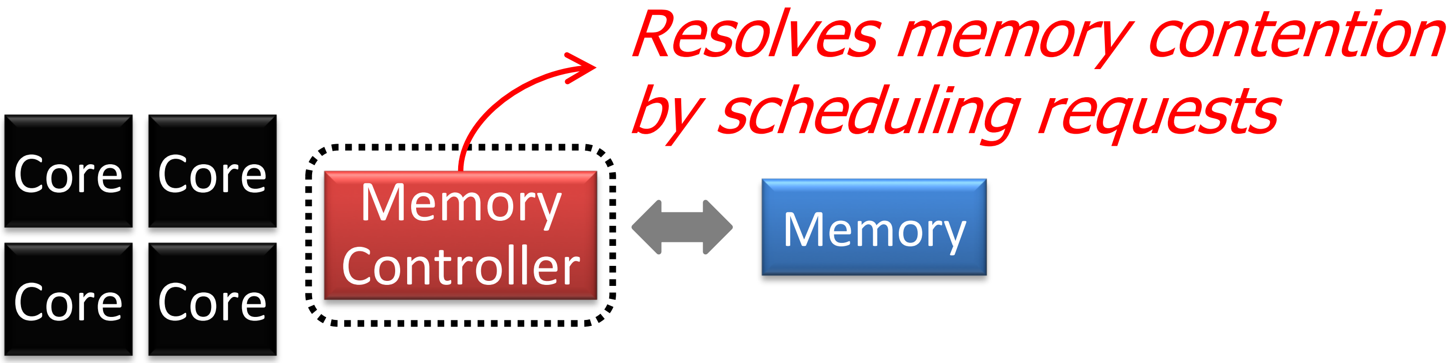
1. Prioritization or request scheduling

2. Data mapping to banks/channels/ranks

3. Core/source throttling

4. Application/thread scheduling

QoS-Aware Memory Scheduling



- How to schedule requests to provide
  - High system performance
  - High fairness to applications
  - Configurability to system software
- Memory controller needs to be aware of threads

QoS-Aware Memory Scheduling: Evolution

QoS-Aware Memory Scheduling: Evolution

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are "heavy" (memory intensive)

- Parallelism-aware batch scheduling [Mutlu+ ISCA'08, Top Picks'09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: Preserving within-thread bank-parallelism improves performance; request batching improves fairness

- ATLAS memory scheduler [Kim+ HPCA'10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: Prioritizing "light" threads improves performance

QoS-Aware Memory Scheduling: Evolution

- Thread cluster memory scheduling [Kim+ MICRO'10, Top Picks'11]
  - Idea: Cluster threads into two groups (latency vs. bandwidth sensitive); prioritize the latency-sensitive ones; employ a fairness policy in the bandwidth sensitive group
  - Takeaway: Heterogeneous scheduling policy that is different based on thread behavior maximizes both performance and fairness

- Integrated Memory Channel Partitioning and Scheduling [Muralidhara+ MICRO'11]
  - Idea: Only prioritize very latency-sensitive threads in the scheduler; mitigate all other applications' interference via channel partitioning
  - Takeaway: Intelligently combining application-aware channel partitioning and memory scheduling provides better performance than either

QoS-Aware Memory Scheduling: Evolution

- Parallel application memory scheduling [Ebrahimi+ MICRO'11]
  - Idea: Identify and prioritize limiter threads of a multithreaded application in the memory scheduler; provide fast and fair progress to non-limiter threads
  - Takeaway: Carefully prioritizing between limiter and non-limiter threads of a parallel application improves performance

- Staged memory scheduling [Ausavarungnirun+ ISCA'12]
  - Idea: Divide the functional tasks of an application-aware memory scheduler into multiple distinct stages, where each stage is significantly simpler than a monolithic scheduler
  - Takeaway: Staging enables the design of a scalable and relatively simpler application-aware memory scheduler that works on very large request buffers

QoS-Aware Memory Scheduling: Evolution

- MISE: Memory Slowdown Model [Subramanian+ HPCA'13]
  - Idea: Estimate the performance of a thread by estimating its change in memory request service rate when run alone vs. shared → use this simple model to estimate slowdown to design a scheduling policy that provides predictable performance or fairness
  - Takeaway: Request service rate of a thread is a good proxy for its performance; alone request service rate can be estimated by giving high priority to the thread in memory scheduling for a while

- ASM: Application Slowdown Model [Subramanian+ MICRO'15]
  - Idea: Extend MISE to take into account cache+memory interference
  - Takeaway: Cache access rate of an application can be estimated accurately and is a good proxy for application performance

QoS-Aware Memory Scheduling: Evolution

- BLISS: Blacklisting Memory Scheduler [Subramanian+ ICCD'14, TPDS'16]
  - Idea: Deprioritize (i.e., blacklist) a thread that has consecutively serviced a large number of requests
  - Takeaway: Blacklisting greatly reduces interference enables the scheduler to be simple without requiring full thread ranking

- DASH: Deadline-Aware Memory Scheduler [Usui+ TACO'16]
  - Idea: Balance prioritization between CPUs, GPUs and Hardware Accelerators (HWA) by keeping HWA progress in check vs. deadlines such that HWAs do not hog performance and appropriately distinguishing between latency-sensitive vs. bandwidth-sensitive CPU workloads
  - Takeaway: Proper control of HWA progress and application-aware CPU prioritization leads to better system performance while meeting HWA deadlines

QoS-Aware Memory Scheduling: Evolution

- Prefetch-aware shared resource management [Ebrahimi+ ISCA'11] [Ebrahimi+ MICRO'09] [Ebrahimi+ HPCA'09] [Lee+ MICRO'08'09]
  - Idea: Prioritize prefetches depending on how they affect system performance; even accurate prefetches can degrade performance of the system
  - Takeaway: Carefully controlling and prioritizing prefetch requests improves performance and fairness

- DRAM-Aware last-level cache policies and write scheduling [Lee+ HPS Tech Report'10] [Seshadri+ ISCA'14]
  - Idea: Design cache eviction and replacement policies such that they proactively exploit the state of the memory controller and DRAM (e.g., proactively evict data from the cache that hit in open rows)
  - Takeaway: Coordination of last-level cache and DRAM policies improves performance and fairness; writes should not be ignored

QoS-Aware Memory Scheduling: Evolution

- FIRM: Memory Scheduling for NVM [Zhao+ MICRO'14]
  - Idea: Carefully handle write-read prioritization with coarse-grained batching and application-aware scheduling
  - Takeaway: Carefully controlling and prioritizing write requests improves performance and fairness; write requests are especially critical in NVMs

- Criticality-Aware Memory Scheduling for GPUs [Jog+ SIGMETRICS'16]
  - Idea: Prioritize latency-critical cores' requests in a GPU system
  - Takeaway: Need to carefully balance locality and criticality to make sure performance improves by taking advantage of both

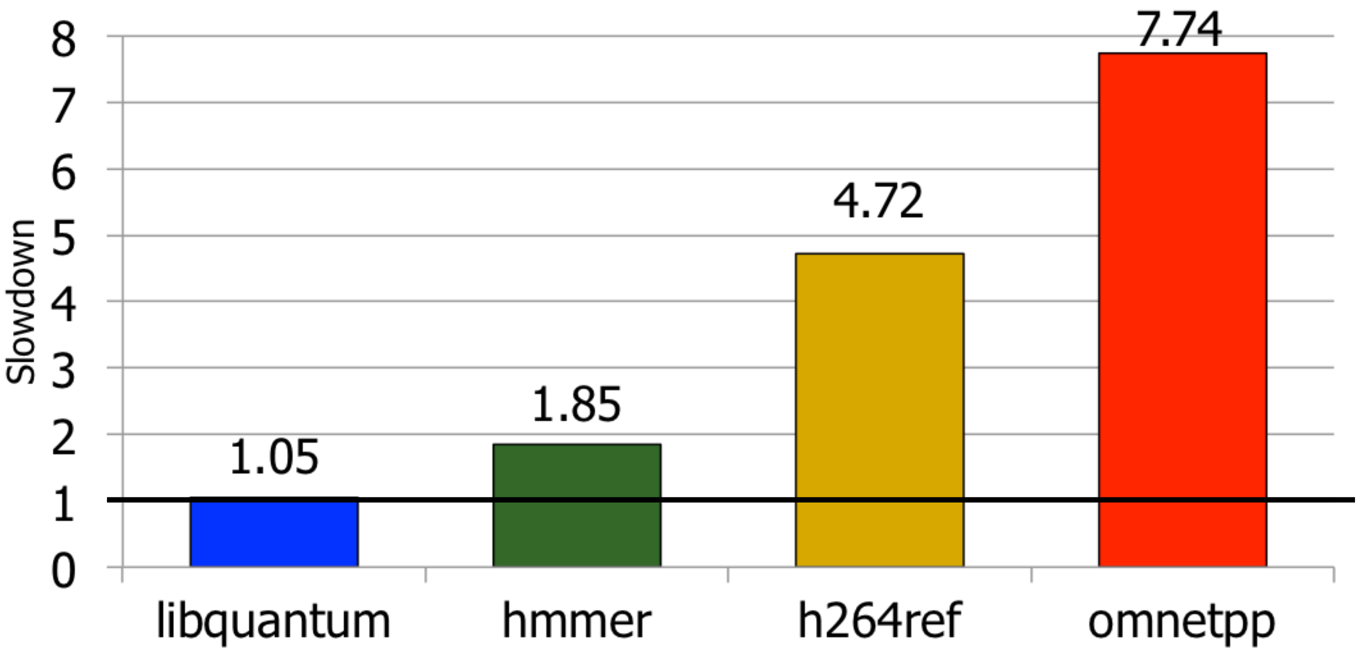
- Worst-case Execution Time Based Memory Scheduling for Real-Time Systems [Kim+ RTAS'14, JRTS'16]

Stall-Time Fair Memory Scheduling

Onur Mutlu and Thomas Moscibroda,
"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"
40th International Symposium on Microarchitecture (***MICRO***),
pages 146-158, Chicago, IL, December 2007. Slides (ppt)

STFM Micro 2007 Talk

The Problem: Unfairness



- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system

How Do We Solve the Problem?

- Stall-time fair memory scheduling [Mutlu+ MICRO'07]
- Goal: Threads sharing main memory should experience similar slowdowns compared to when they are run alone → fair scheduling
    - Also improves overall system performance by ensuring cores make "proportional" progress
- Idea: Memory controller estimates each thread's slowdown due to interference and schedules requests in a way to balance the slowdowns
- Mutlu and Moscibroda, "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors," MICRO 2007.

Stall-Time Fairness in Shared DRAM Systems

- A DRAM system is fair if it equalizes the slowdown of equal-priority threads relative to when each thread is run alone on the same system

- DRAM-related stall-time: The time a thread spends waiting for DRAM memory
- ST_{shared}: DRAM-related stall-time when the thread runs with other threads
- ST_{alone}: DRAM-related stall-time when the thread runs alone
- **Memory-slowdown = ST_{shared}/ST_{alone}**
  - Relative increase in stall-time

- *Stall-Time Fair Memory scheduler (STFM)* aims to equalize Memory-slowdown for interfering threads, without sacrificing performance
  - Considers inherent DRAM performance of each thread
  - Aims to allow proportional progress of threads

STFM Scheduling Algorithm [MICRO' 07]

- For each thread, the DRAM controller
  - Tracks ST_{shared}
  - Estimates ST_{alone}

- Each cycle, the DRAM controller
  - Computes Slowdown = ST_{shared}/ST_{alone} for threads with legal requests
  - Computes unfairness = MAX Slowdown / MIN Slowdown

- If unfairness $< \alpha$
  - Use DRAM throughput oriented scheduling policy
- If unfairness $\geq \alpha$
  - Use fairness-oriented scheduling policy
    - (1) requests from thread with MAX Slowdown first
    - (2) row-hit first , (3) oldest-first

How Does STFM Prevent Unfairness?

T0: Row 0

T1: Row 5

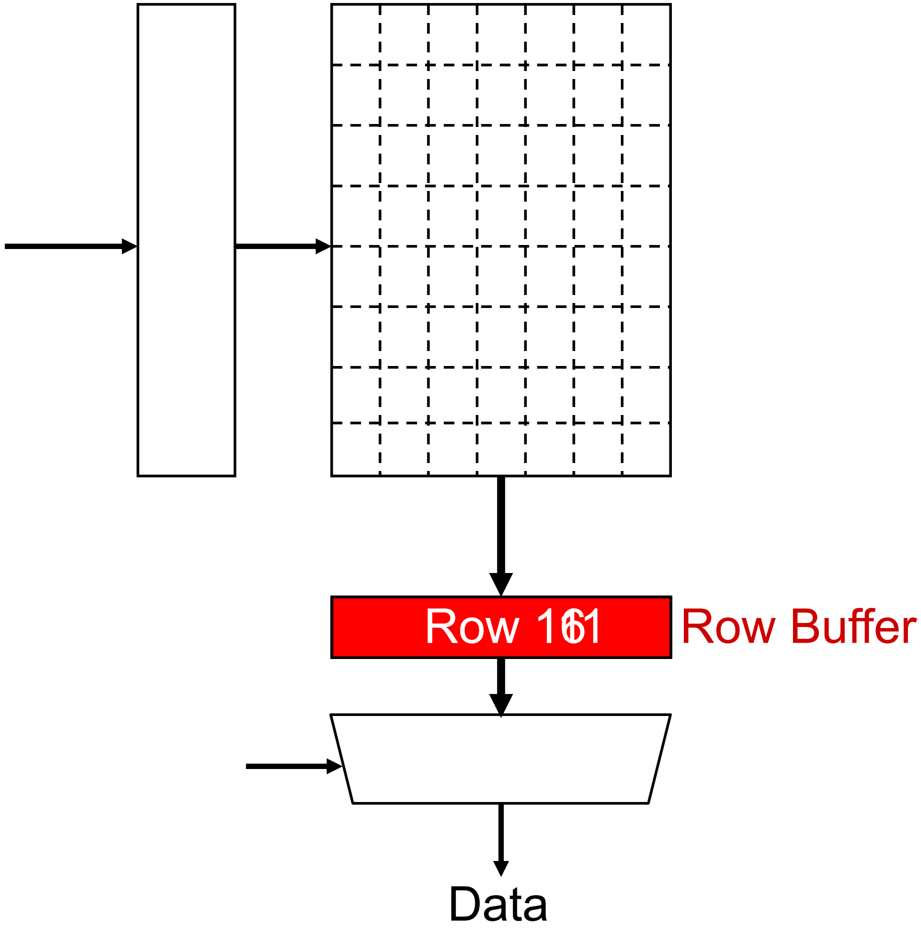
T0: Row 0

T1: Row 111

T0: Row 0

T1: Row 16
T0: Row 0

T0 Slowdown	1.04
T1 Slowdown	1.06
Unfairness	1.06
α	1.05



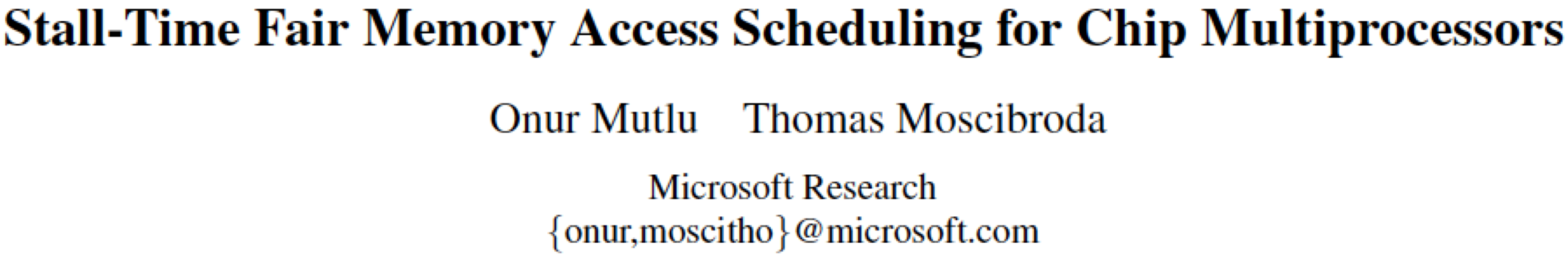
STFM Pros and Cons

- Upsides:
  - First algorithm for fair multi-core memory scheduling
  - Provides a mechanism to estimate memory slowdown of a thread
  - Good at providing fairness
  - Being fair can improve performance

- Downsides:
  - Does not handle all types of interference
  - (Somewhat) complex to implement
  - Slowdown estimations can be incorrect

More on STFM

- Onur Mutlu and Thomas Moscibroda, **"Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors"** *Proceedings of the 40th International Symposium on Microarchitecture (**MICRO**)*, pages 146-158, Chicago, IL, December 2007. [Summary] [Slides (ppt)]



Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors

Onur Mutlu   Thomas Moscibroda

Microsoft Research
{onur,moscitho}@microsoft.com

Parallelism-Aware Batch Scheduling

Onur Mutlu and Thomas Moscibroda,
"Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems"
35th International Symposium on Computer Architecture (***ISCA***),
pages 63-74, Beijing, China, June 2008. Slides (ppt)

PAR-BS ISCA 2008 Talk

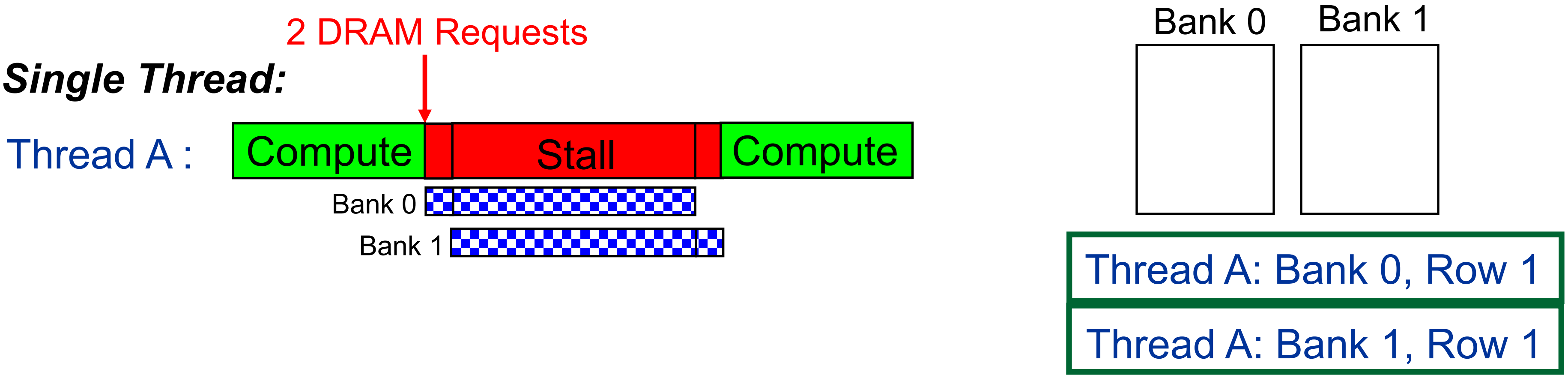
Another Problem due to Memory Interference

- Processors try to tolerate the latency of DRAM requests by generating multiple outstanding requests
  - Memory-Level Parallelism (MLP)
  - Out-of-order execution, non-blocking caches, runahead execution

- Effective only if the DRAM controller actually services the multiple requests in parallel in DRAM banks

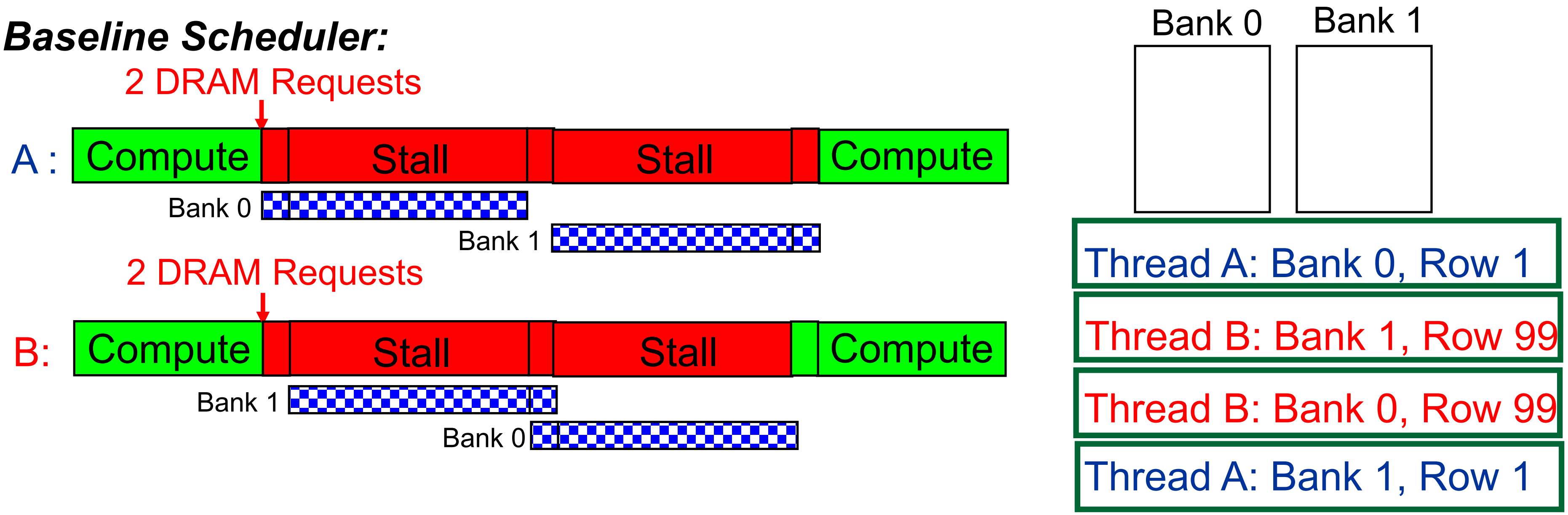
- Multiple threads share the DRAM controller
- DRAM controllers are not aware of a thread's MLP
  - Can service each thread's outstanding requests serially, not in parallel

Bank Parallelism of a Thread



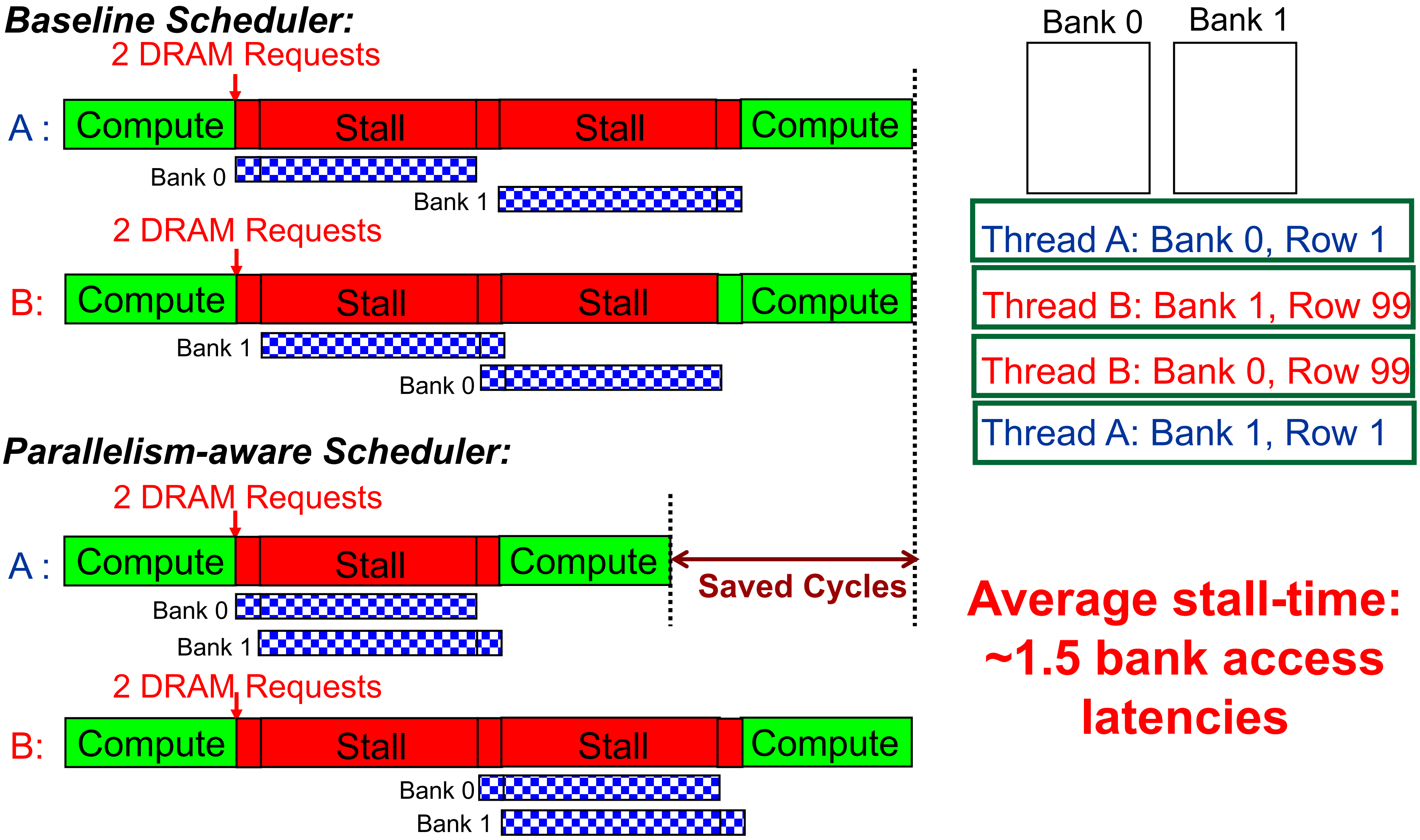
Bank access latencies of the two requests overlapped
Thread stalls for ~ONE bank access latency

Bank Parallelism Interference in DRAM



Bank access latencies of each thread serialized
Each thread stalls for ~TWO bank access latencies

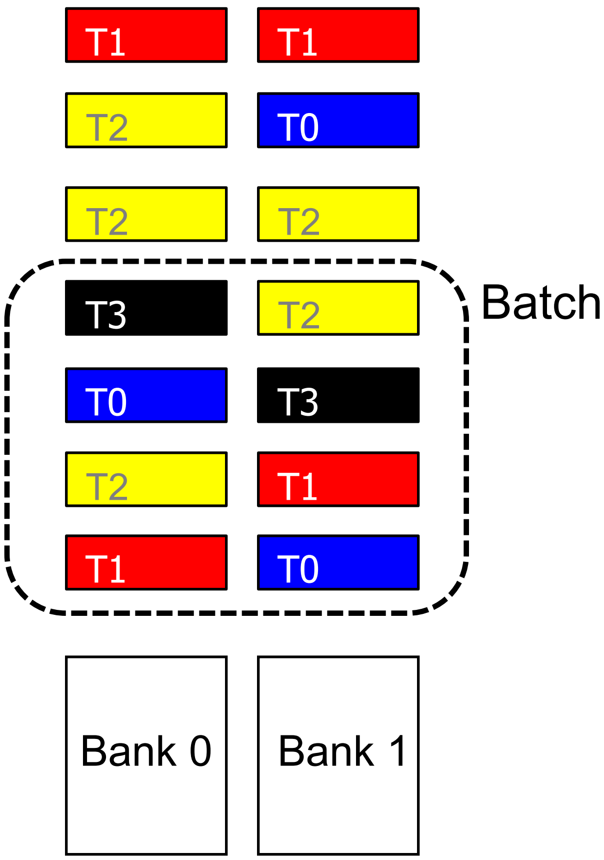
Parallelism-Aware Scheduler



Average stall-time: ~1.5 bank access latencies

Parallelism-Aware Batch Scheduling (PAR-BS)

- Principle 1: Parallelism-awareness
  - Schedule requests from a thread (to different banks) back to back
  - Preserves each thread's bank parallelism
  - But, this can cause starvation…



- Principle 2: Request Batching
  - Group a fixed number of oldest requests from each thread into a "batch"
  - Service the batch before all other requests
  - Form a new batch when the current one is done
  - Eliminates starvation, provides fairness
  - Allows parallelism-awareness within a batch

Mutlu and Moscibroda, "Parallelism-Aware Batch Scheduling," ISCA 2008.

PAR-BS Components

- Request batching

- Within-batch scheduling
  - Parallelism aware

Request Batching

- Each memory request has a bit (*marked)* associated with it

- Batch formation:
  - Mark up to *Marking-Cap* oldest requests per bank for each thread
  - Marked requests constitute the batch
  - Form a new batch when no marked requests are left

- Marked requests are prioritized over unmarked ones
  - No reordering of requests across batches: no starvation, high fairness

- How to prioritize requests within a batch?

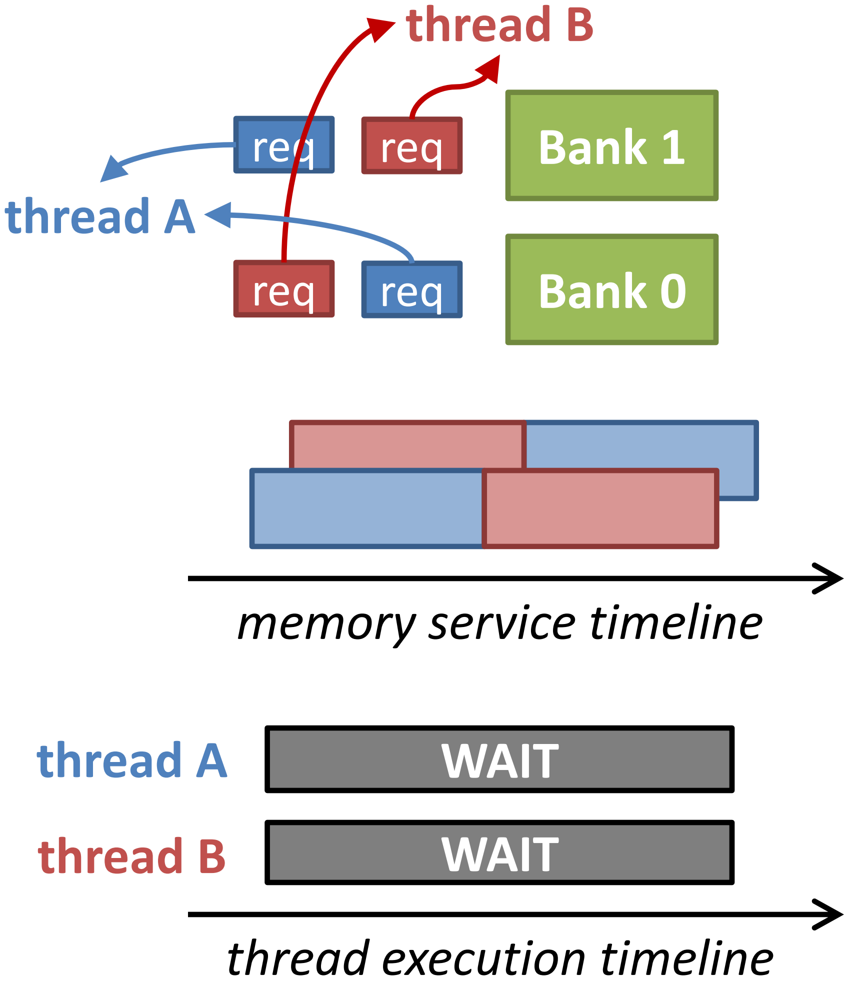
Within-Batch Scheduling

- Can use any existing DRAM scheduling policy
  - FR-FCFS (row-hit first, then oldest-first) exploits row-buffer locality
- But, we also want to preserve intra-thread bank parallelism
  - Service each thread’s requests back to back

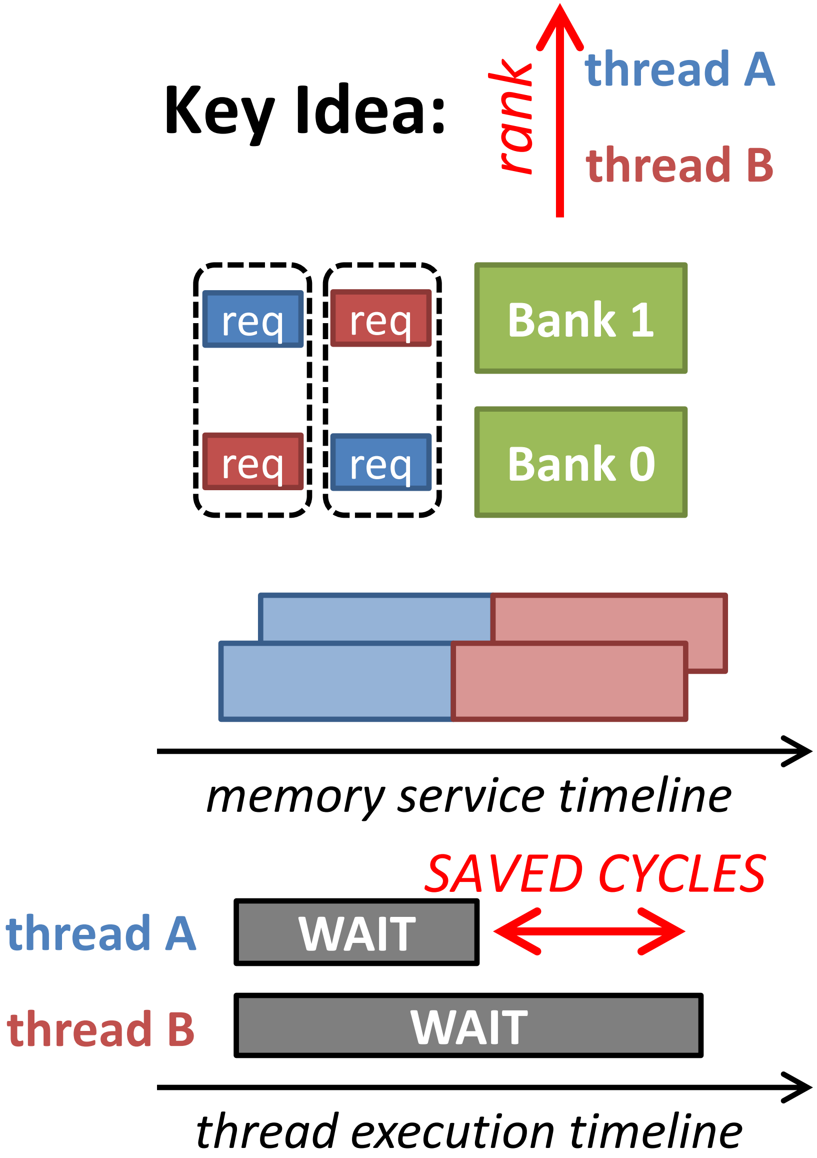
HOW?

- Scheduler computes a ranking of threads when the batch is formed
  - Higher-ranked threads are prioritized over lower-ranked ones
  - Improves the likelihood that requests from a thread are serviced in parallel by different banks
    - Different threads prioritized in the same order across ALL banks

Thread Ranking



Key Idea:



How to Rank Threads within a Batch

- Ranking scheme affects system throughput and fairness

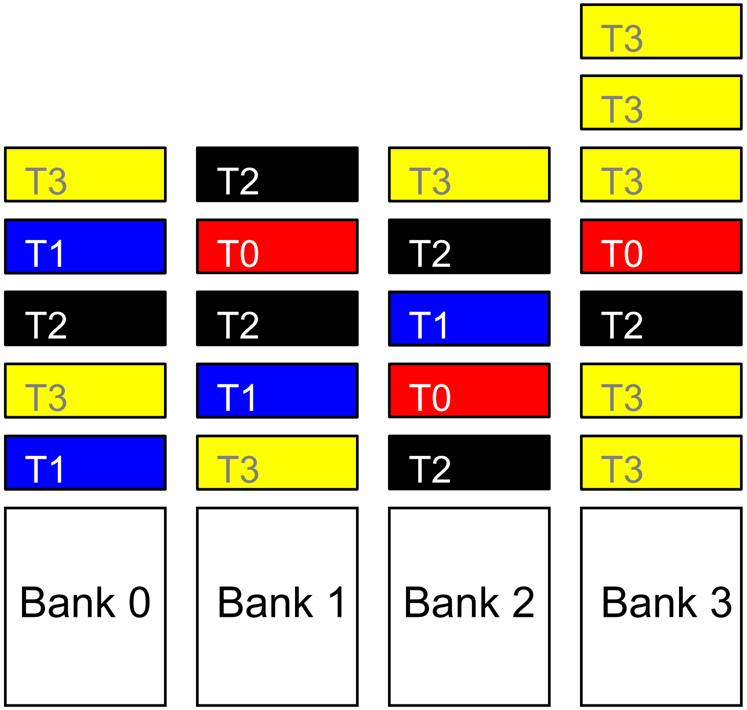
- Maximize system throughput
  - Minimize average stall-time of threads within the batch
- Minimize unfairness (Equalize the slowdown of threads)
  - Service threads with inherently low stall-time early in the batch
  - Insight: delaying memory non-intensive threads results in high slowdown

- Shortest stall-time first (shortest job first) ranking
  - Provides optimal system throughput [Smith, 1956]*
  - Controller estimates each thread's stall-time within the batch
  - Ranks threads with shorter stall-time higher

* W.E. Smith, "Various optimizers for single stage production," Naval Research Logistics Quarterly, 1956.

Shortest Stall-Time First Ranking

- Maximum number of marked requests to any bank (max-bank-load)
  - Rank thread with lower max-bank-load higher (~ low stall-time)
- Total number of marked requests (total-load)
  - Breaks ties: rank thread with lower total-load higher

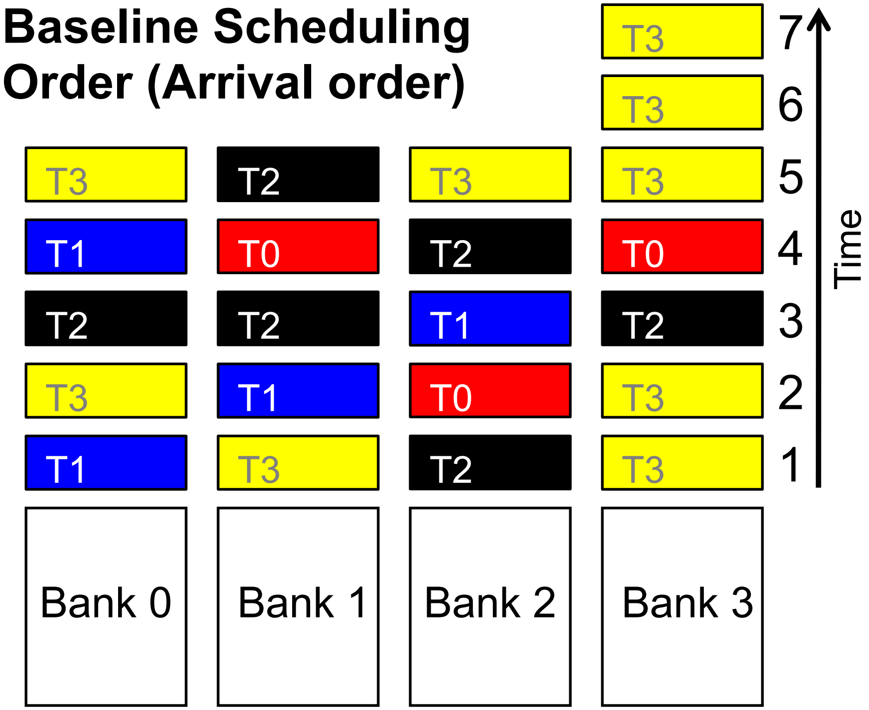


	max-bank-load	total-load

Ranking:
T0 > T1 > T2 > T3

Example Within-Batch Scheduling Order

Baseline Scheduling Order (Arrival order)

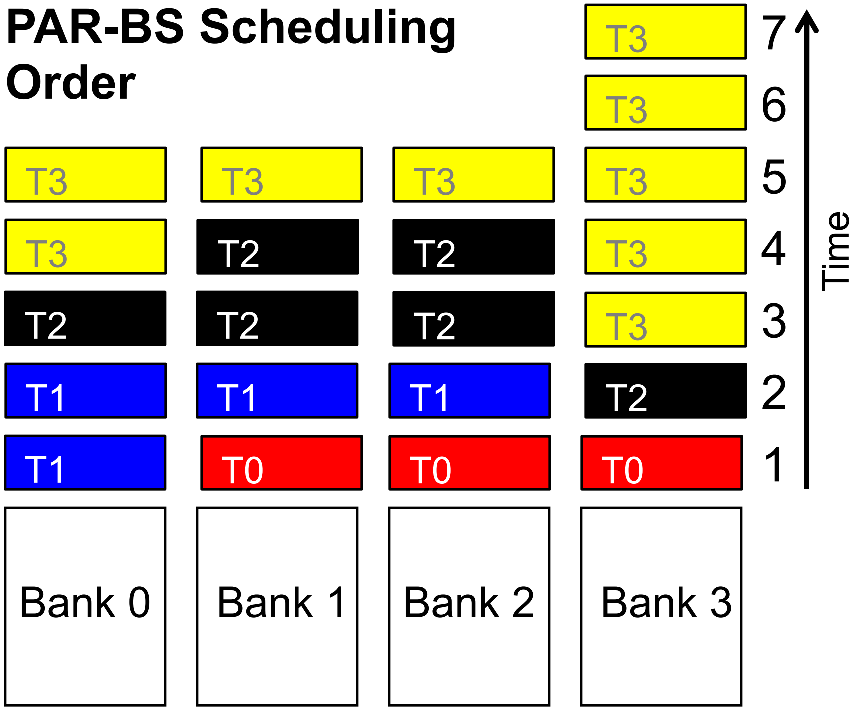


Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T2	T3	T3
4	T1	T0	T2	T0
3	T2	T2	T1	T2
2	T3	T1	T0	T3
1	T1	T3	T2	T3

	T0	T1	T2	T3
Stall times				

AVG: 5 bank access latencies

PAR-BS Scheduling Order



Time	Bank 0	Bank 1	Bank 2	Bank 3
7				T3
6				T3
5	T3	T3	T3	T3
4	T3	T2	T2	T3
3	T2	T2	T2	T3
2	T1	T1	T1	T2
1	T1	T0	T0	T0

Ranking: T0 > T1 > T2 > T3

	T0	T1	T2	T3
Stall times				

AVG: 3.5 bank access latencies

Putting It Together: PAR-BS Scheduling Policy

- PAR-BS Scheduling Policy

  (1) Marked requests first

  Batching

  (2) Row-hit requests first

  (3) Higher-rank thread first (shortest stall-time first)

  (4) Oldest first

  Parallelism-aware within-batch scheduling

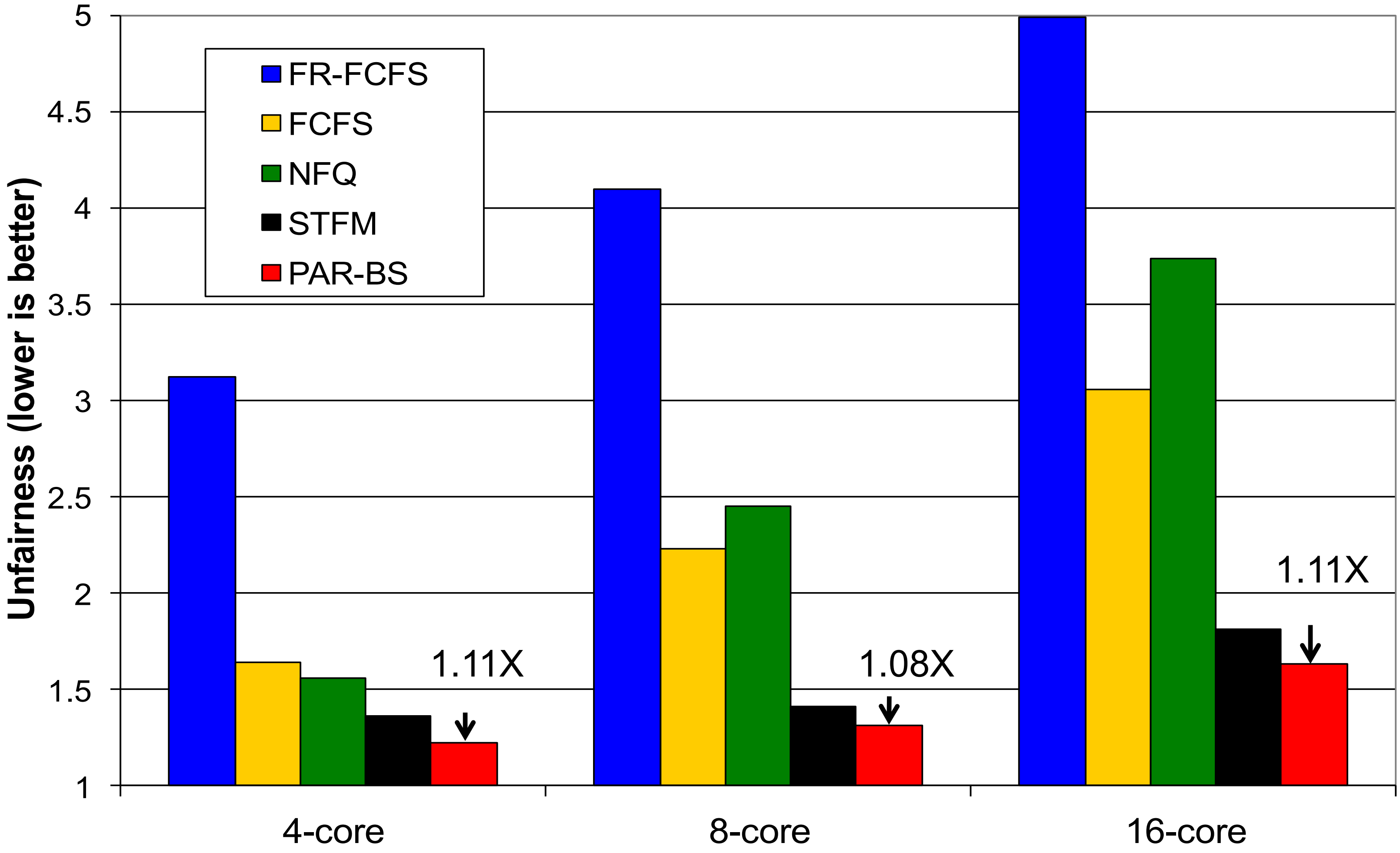
- Three properties:
  - Exploits row-buffer locality **and** intra-thread bank parallelism
  - Work-conserving
    - Services unmarked requests to banks without marked requests
  - Marking-Cap is important
    - Too small cap: destroys row-buffer locality
    - Too large cap: penalizes memory non-intensive threads
- Many more trade-offs analyzed in the paper

Hardware Cost

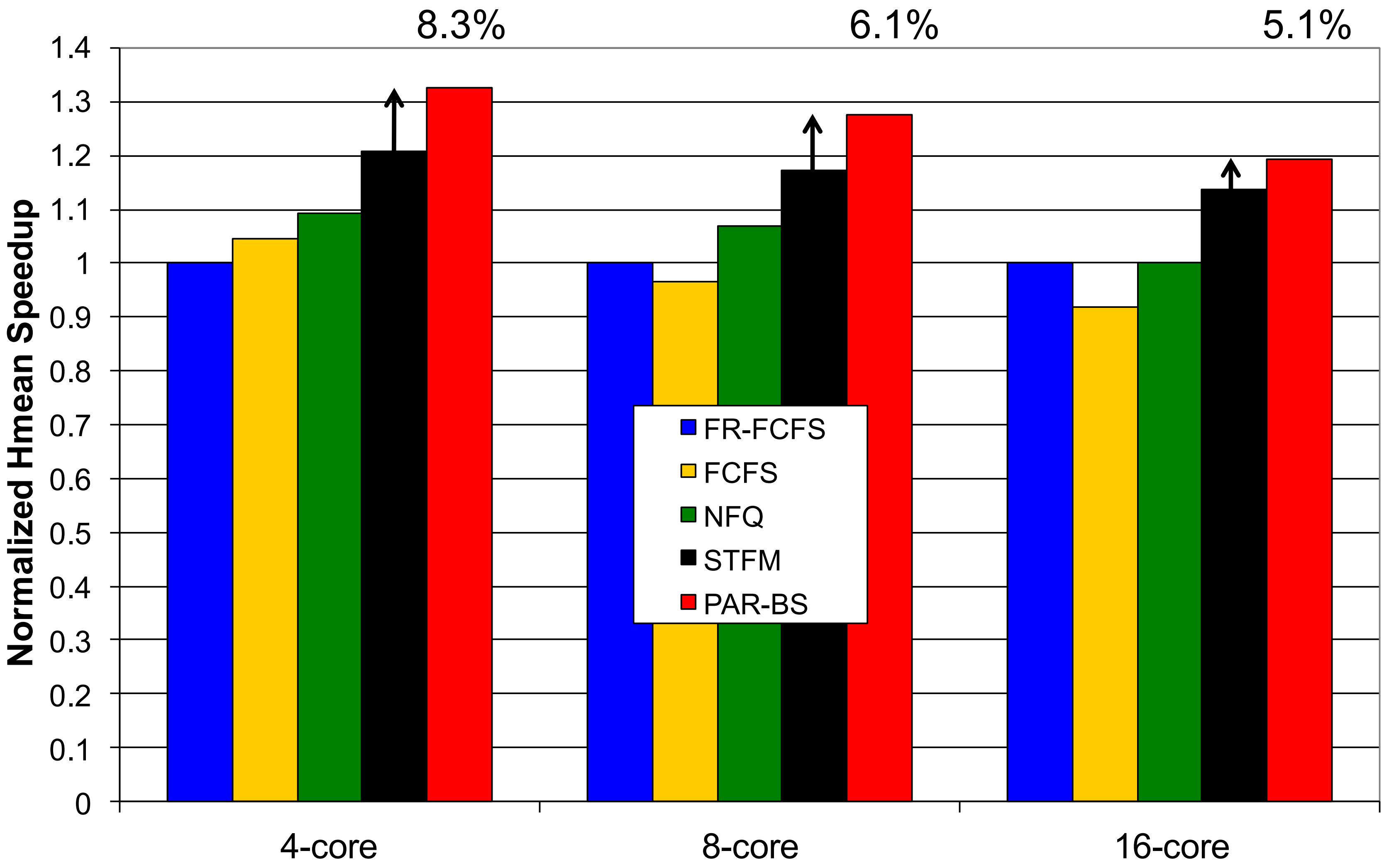
- <1.5KB storage cost for
  - 8-core system with 128-entry memory request buffer
- No complex operations (e.g., divisions)
- Not on the critical path
  - Scheduler makes a decision only every DRAM cycle

Unfairness on 4-, 8-, 16-core Systems

Unfairness = MAX Memory Slowdown / MIN Memory Slowdown [MICRO 2007]



System Performance (Hmean-speedup)



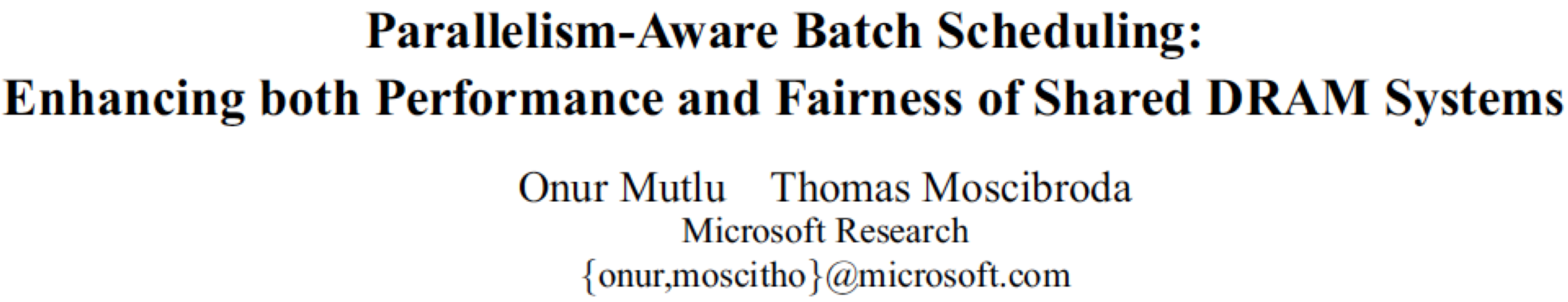
PAR-BS Pros and Cons

- Upsides:
  - First scheduler to address bank parallelism destruction across multiple threads
  - Simple mechanism (vs. STFM)
  - Batching provides fairness
  - Ranking enables parallelism awareness

- Downsides:
  - Does not always prioritize the latency-sensitive applications

More on PAR-BS

- Onur Mutlu and Thomas Moscibroda, **"Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems"** *Proceedings of the 35th International Symposium on Computer Architecture (**ISCA**)*, pages 63-74, Beijing, China, June 2008. [Summary] [Slides (ppt)]
  One of the 12 computer architecture papers of 2008 selected as Top Picks by IEEE Micro.



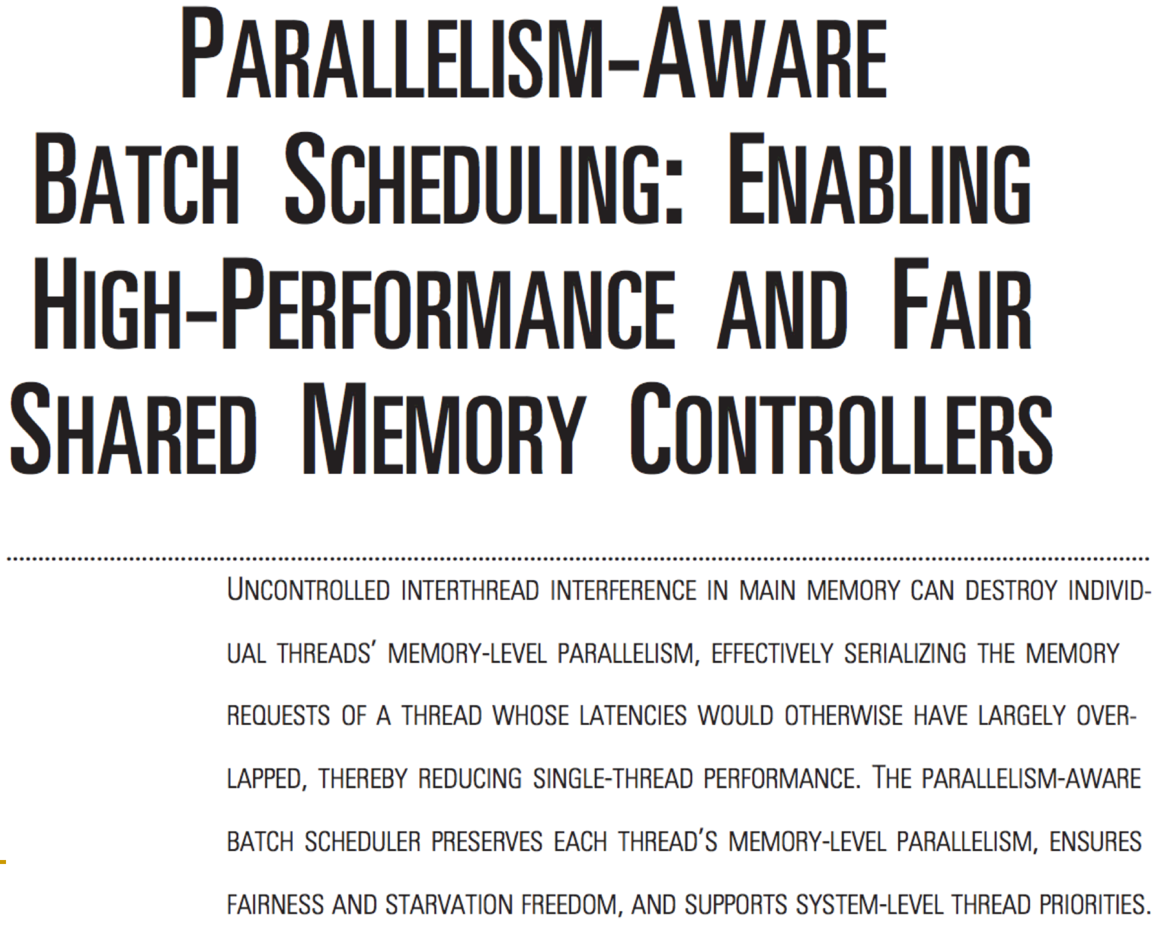
Parallelism-Aware Batch Scheduling:
Enhancing both Performance and Fairness of Shared DRAM Systems

Onur Mutlu Thomas Moscibroda
Microsoft Research
{onur,moscitho}@microsoft.com

http://www.youtube.com/watch?v=UB1kgYR-4V0

More on PAR-BS

- Onur Mutlu and Thomas Moscibroda, **"Parallelism-Aware Batch Scheduling: Enabling High-Performance and Fair Memory Controllers"** *IEEE Micro, Special Issue: Micro's Top Picks from 2008 Computer Architecture Conferences (**MICRO TOP PICKS**)*, Vol. 29, No. 1, pages 22-32, January/February 2009.



PARALLELISM-AWARE BATCH SCHEDULING: ENABLING HIGH-PERFORMANCE AND FAIR SHARED MEMORY CONTROLLERS

UNCONTROLLED INTERTHREAD INTERFERENCE IN MAIN MEMORY CAN DESTROY INDIVIDUAL THREADS' MEMORY-LEVEL PARALLELISM, EFFECTIVELY SERIALIZING THE MEMORY REQUESTS OF A THREAD WHOSE LATENCIES WOULD OTHERWISE HAVE LARGELY OVERLAPPED, THEREBY REDUCING SINGLE-THREAD PERFORMANCE. THE PARALLELISM-AWARE BATCH SCHEDULER PRESERVES EACH THREAD'S MEMORY-LEVEL PARALLELISM, ENSURES FAIRNESS AND STARVATION FREEDOM, AND SUPPORTS SYSTEM-LEVEL THREAD PRIORITIES.

ATLAS Memory Scheduler

Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter,
"ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers"
16th International Symposium on High-Performance Computer Architecture (***HPCA***),
Bangalore, India, January 2010. Slides (pptx)

ATLAS HPCA 2010 Talk

ATLAS: Summary

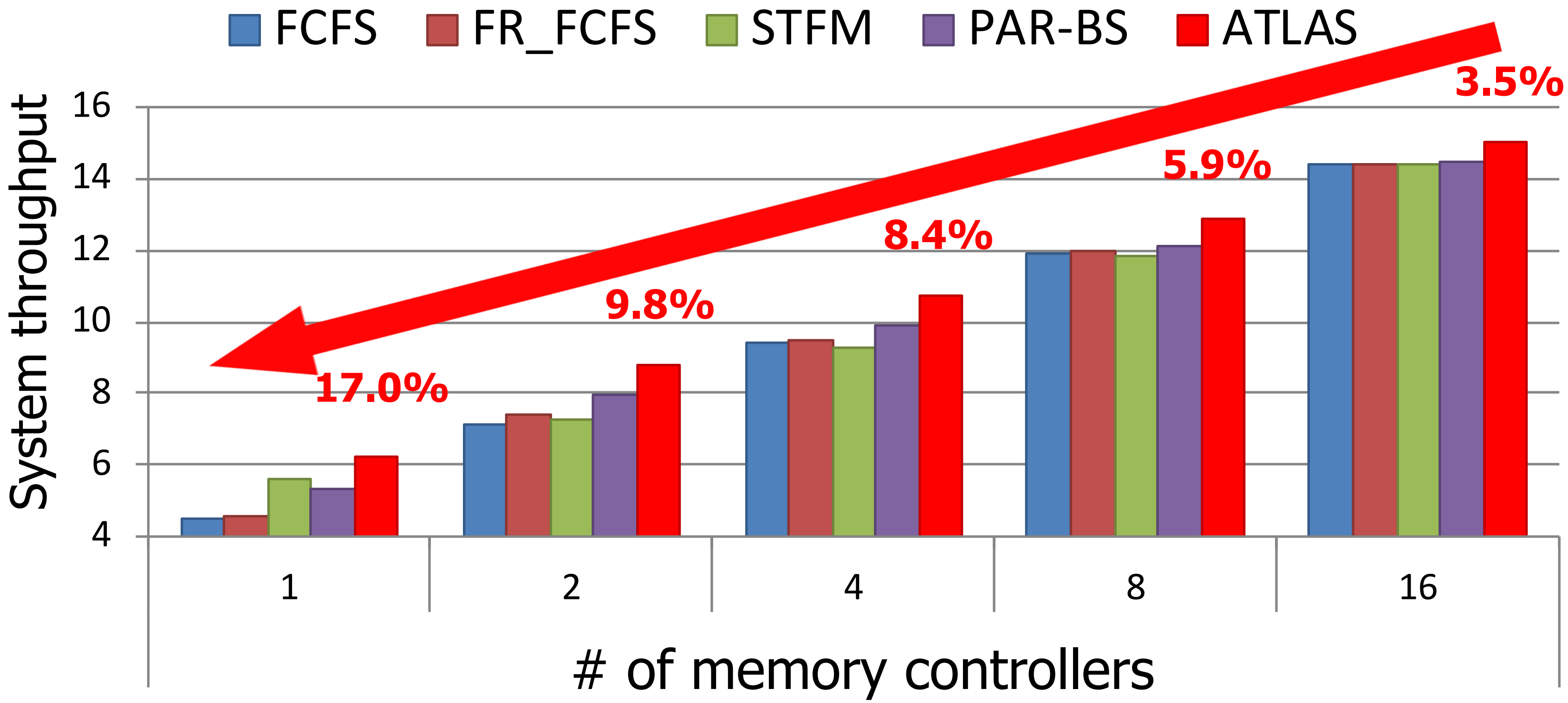
- Goal: To maximize system performance

- Main idea: Prioritize the thread that has attained the least service from the memory controllers (Adaptive per-Thread Least Attained Service Scheduling)
    - Rank threads based on attained service in the past time interval(s)
    - Enforce thread ranking in the memory scheduler during the current interval

- Why it works: Prioritizes "light" (memory non-intensive) threads that are more likely to keep their cores busy

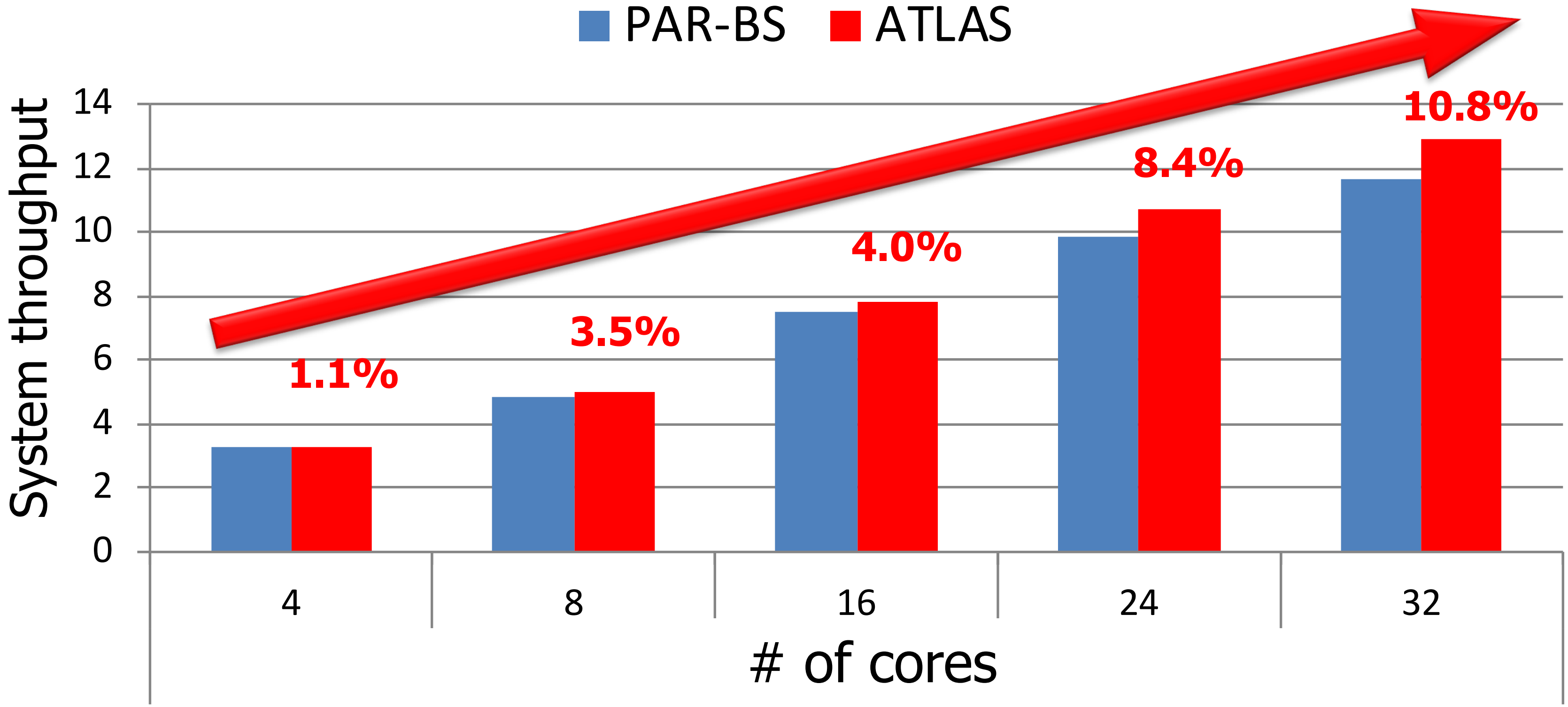
System Throughput: 24-Core System

System throughput = ∑ Speedup



ATLAS consistently provides higher system throughput than all previous scheduling algorithms

System Throughput: 4-MC System



# of cores increases → ATLAS performance benefit increases

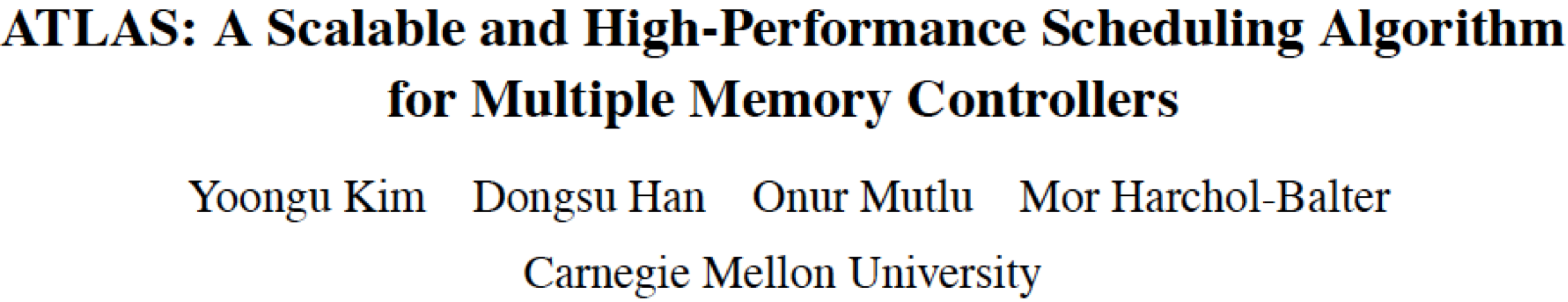
ATLAS Pros and Cons

- Upsides:
  - Good at improving overall throughput (compute-intensive threads are prioritized)
  - Low complexity
  - Coordination among controllers happens infrequently

- Downsides:
  - Lowest/medium ranked threads get delayed significantly → high unfairness

More on ATLAS Memory Scheduler

- Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter, **"ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers"** *Proceedings of the 16th International Symposium on High-Performance Computer Architecture* (***HPCA***), Bangalore, India, January 2010. Slides (pptx)



ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers

Yoongu Kim Dongsu Han Onur Mutlu Mor Harchol-Balter

Carnegie Mellon University

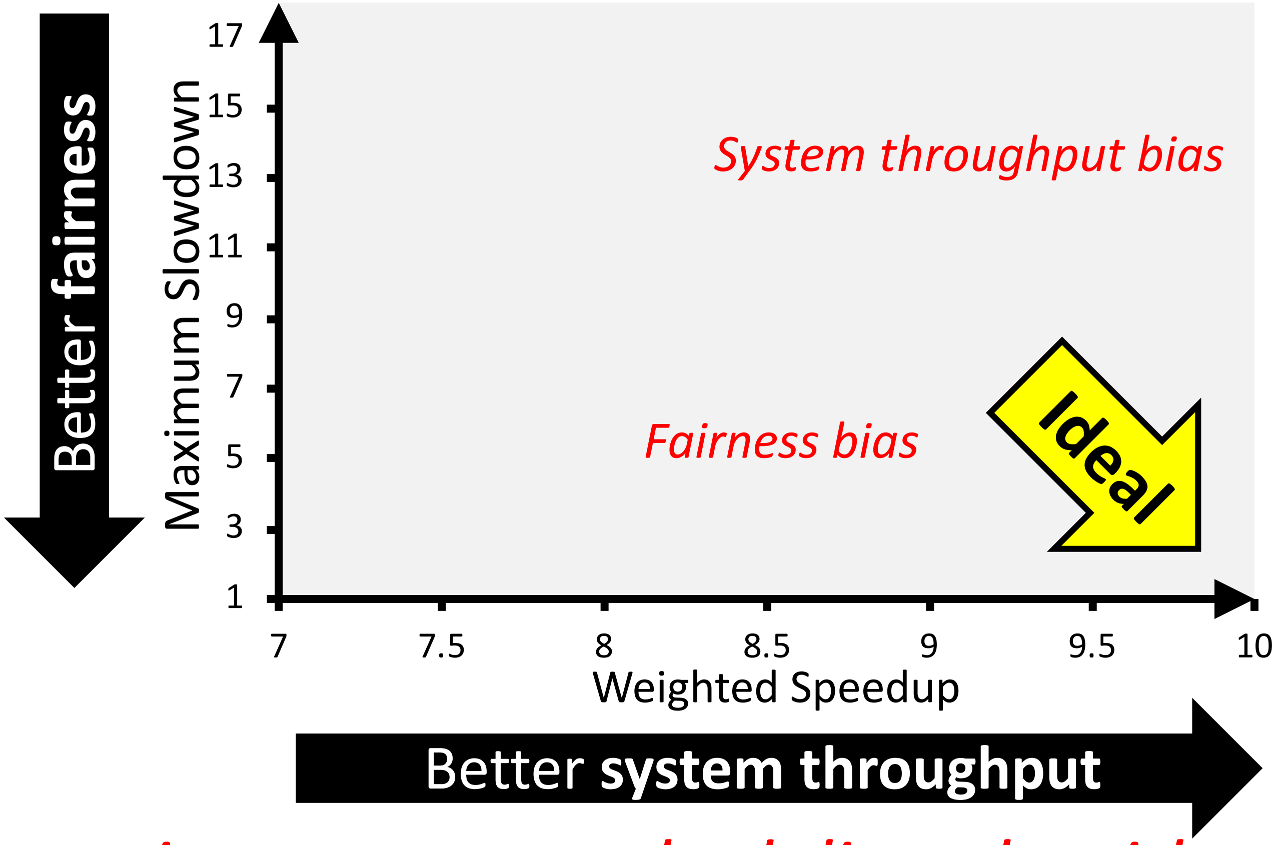
TCM:
Thread Cluster Memory Scheduling

Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter,
"Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior"
43rd International Symposium on Microarchitecture (***MICRO***),
pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)

TCM Micro 2010 Talk

Previous Scheduling Algorithms are Biased

24 cores, 4 memory controllers, 96 workloads



No previous memory scheduling algorithm provides both the best fairness and system throughput

Throughput vs. Fairness

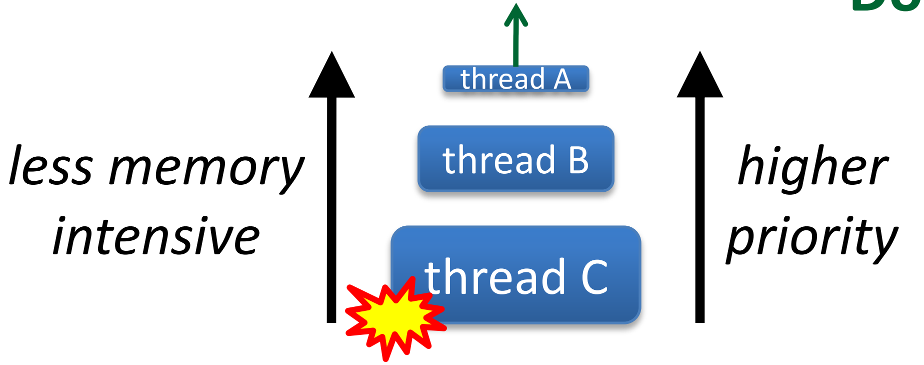
***Throughput biased** approach*

Prioritize less memory-intensive threads

***Fairness biased** approach*

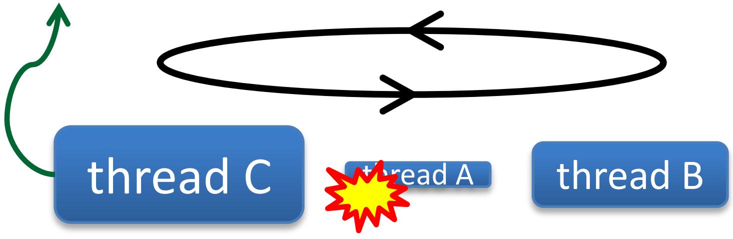
Take turns accessing memory

Good for throughput



starvation → unfairness

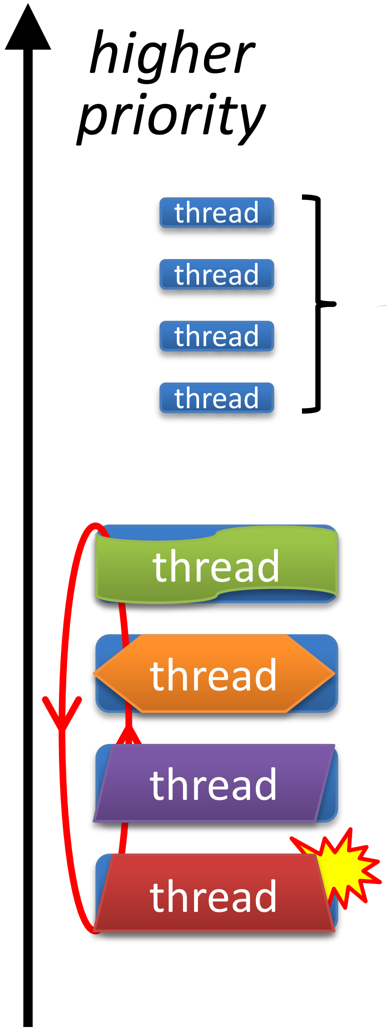
Does not starve



not prioritized → reduced throughput

Single policy for all threads is insufficient

Achieving the Best of Both Worlds



For Throughput

Prioritize memory-non-intensive threads

For Fairness

Unfairness caused by memory-intensive being prioritized over each other

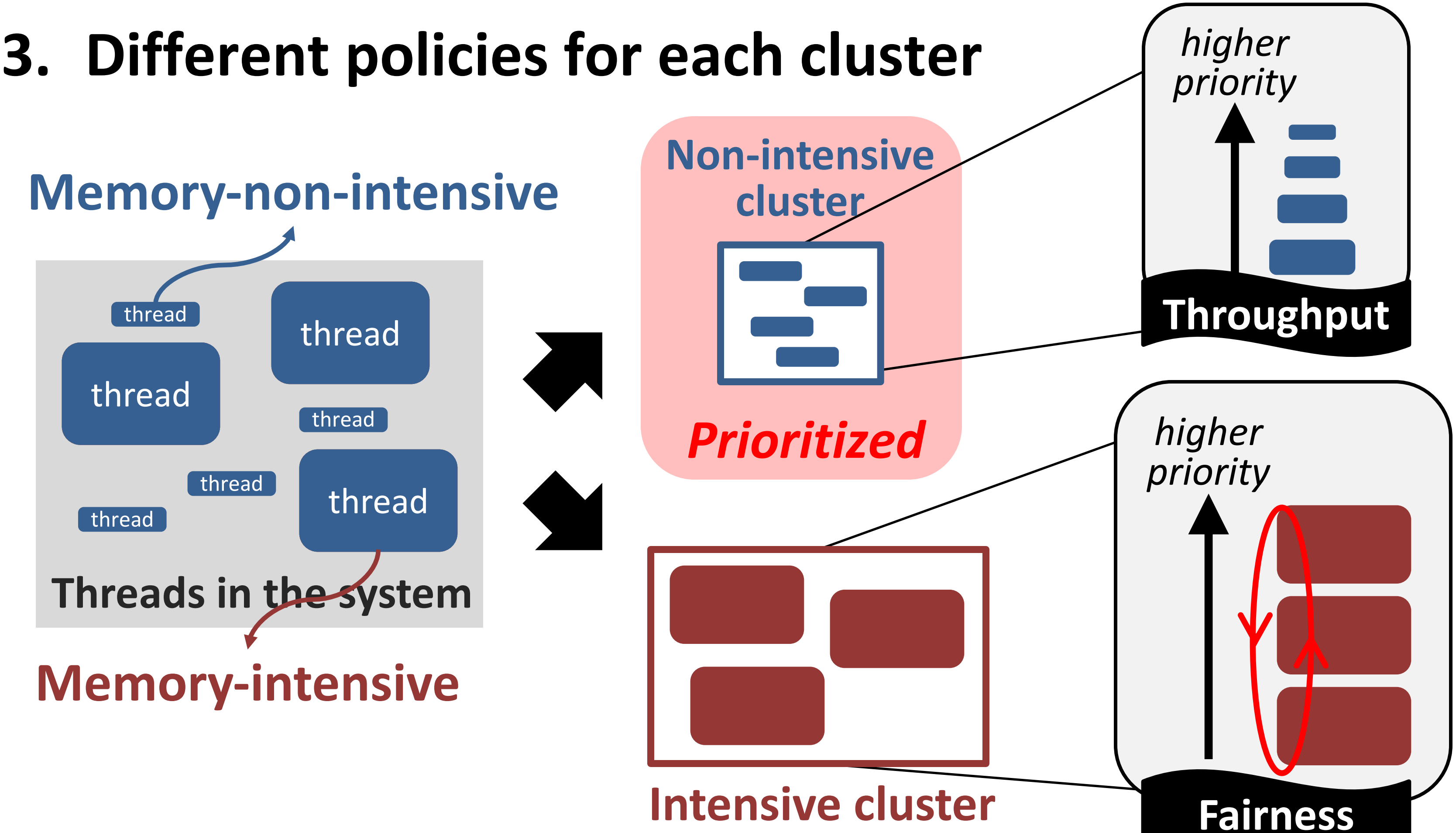
- Shuffle thread ranking

Memory-intensive threads have different vulnerability to interference

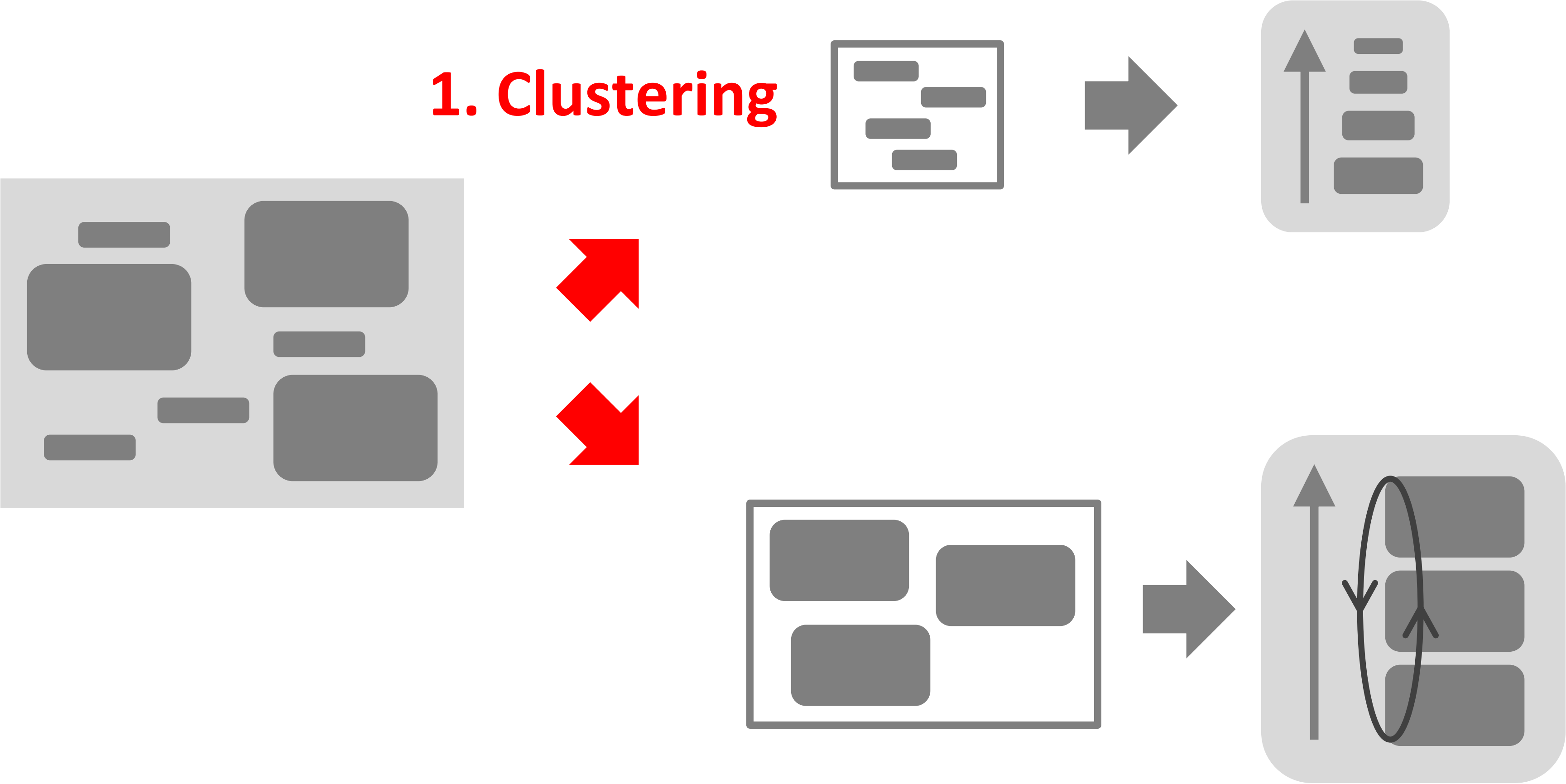
- Shuffle <u>asymmetrically</u>

Thread Cluster Memory Scheduling [Kim+ MICRO'10]

1. Group threads into two ***clusters***
2. Prioritize non-intensive cluster
3. Different policies for each cluster

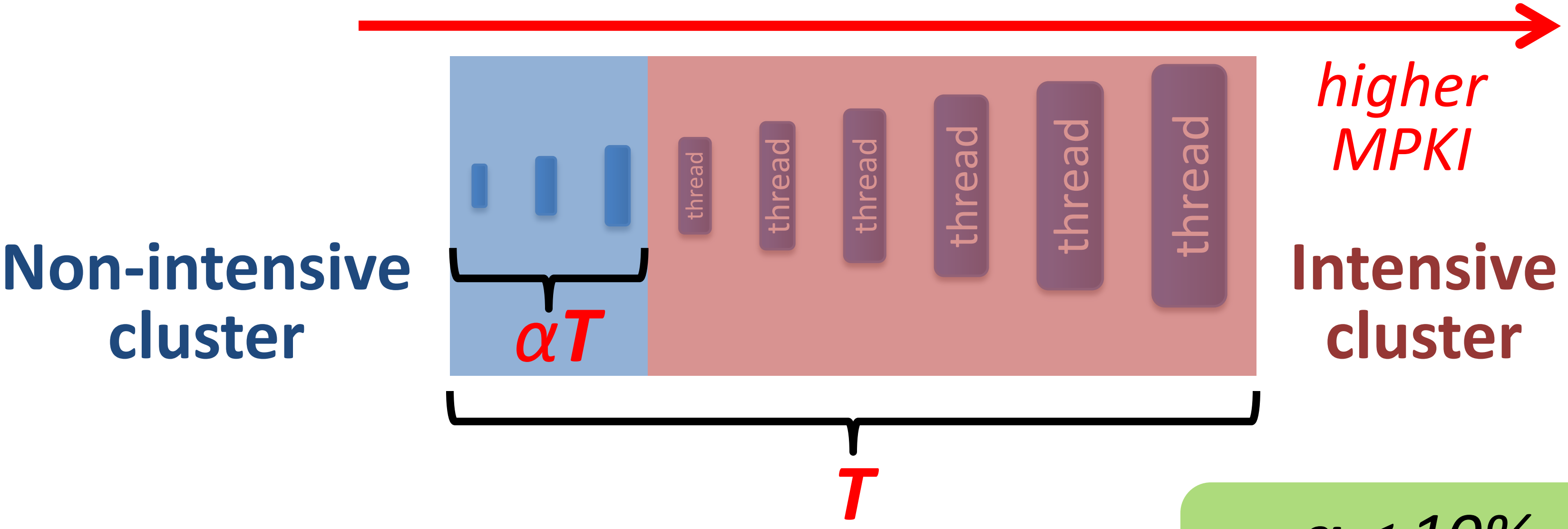


TCM Outline



Clustering Threads

Step1 Sort threads by **MPKI** (misses per kiloinstruction)

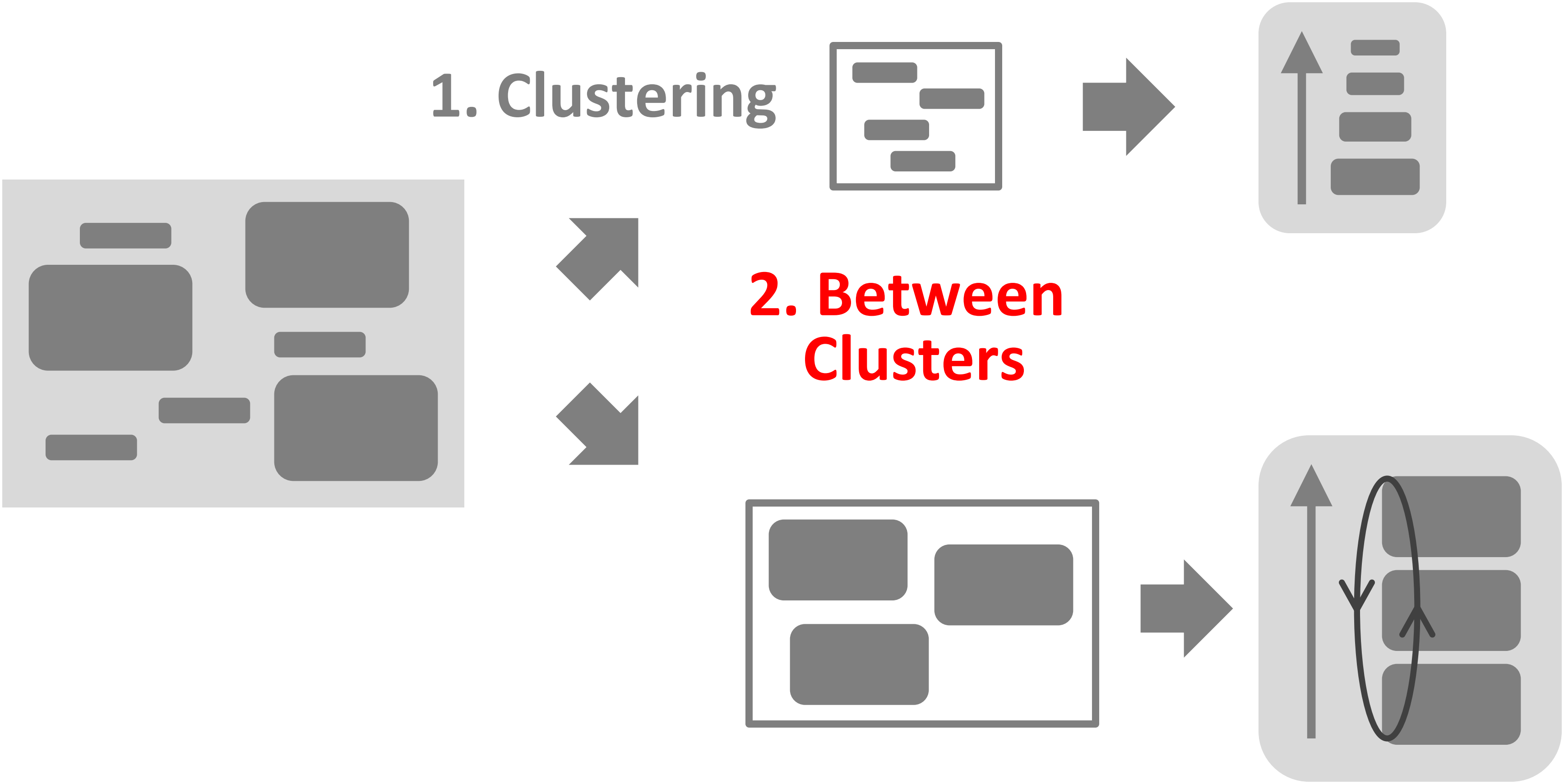


T = Total ***memory bandwidth usage***

***α** < 10%*
ClusterThreshold

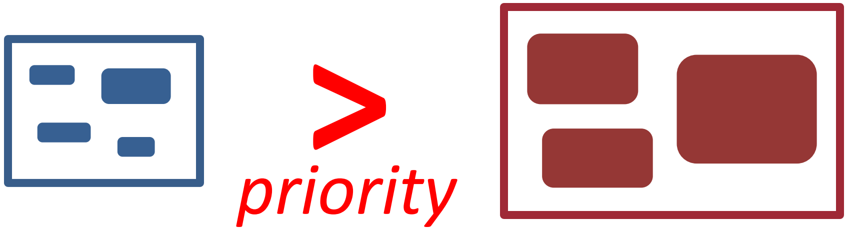
Step2 Memory bandwidth usage *αT* divides clusters

TCM Outline



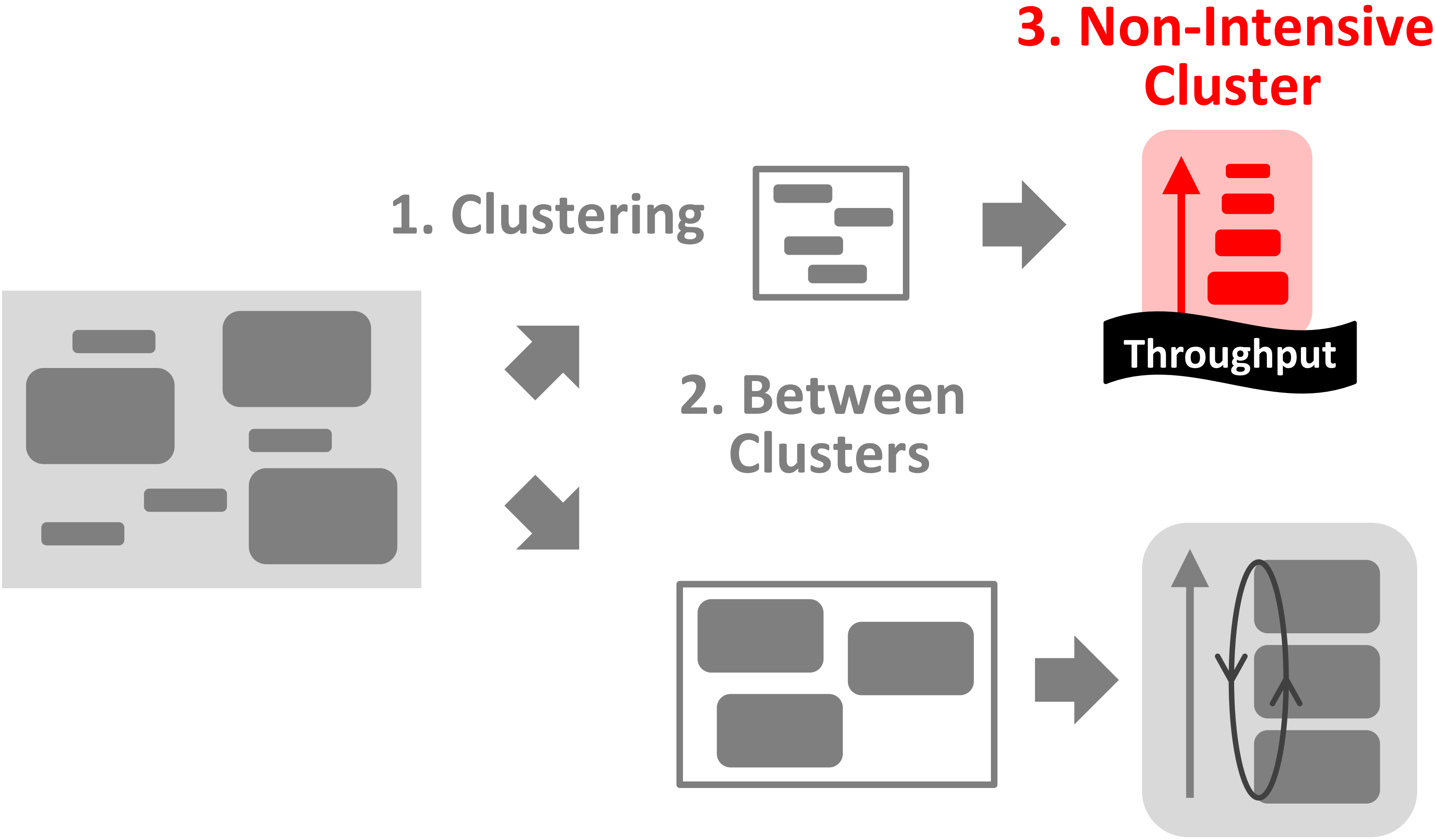
Prioritization Between Clusters

Prioritize non-intensive cluster



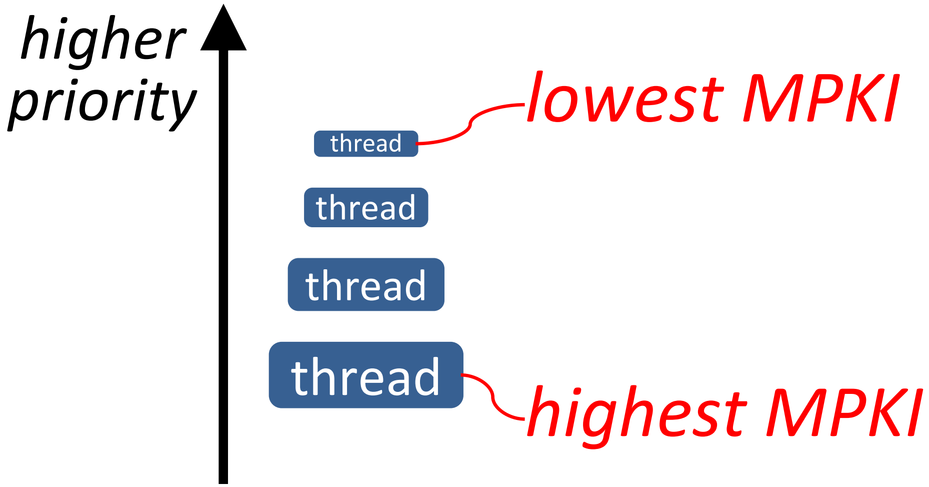
- **Increases system throughput**
  - Non-intensive threads have greater potential for making progress
- **Does not degrade fairness**
  - Non-intensive threads are “light”
  - Rarely interfere with intensive threads

TCM Outline


3. Non-Intensive Cluster
1. Clustering
Throughput
2. Between Clusters

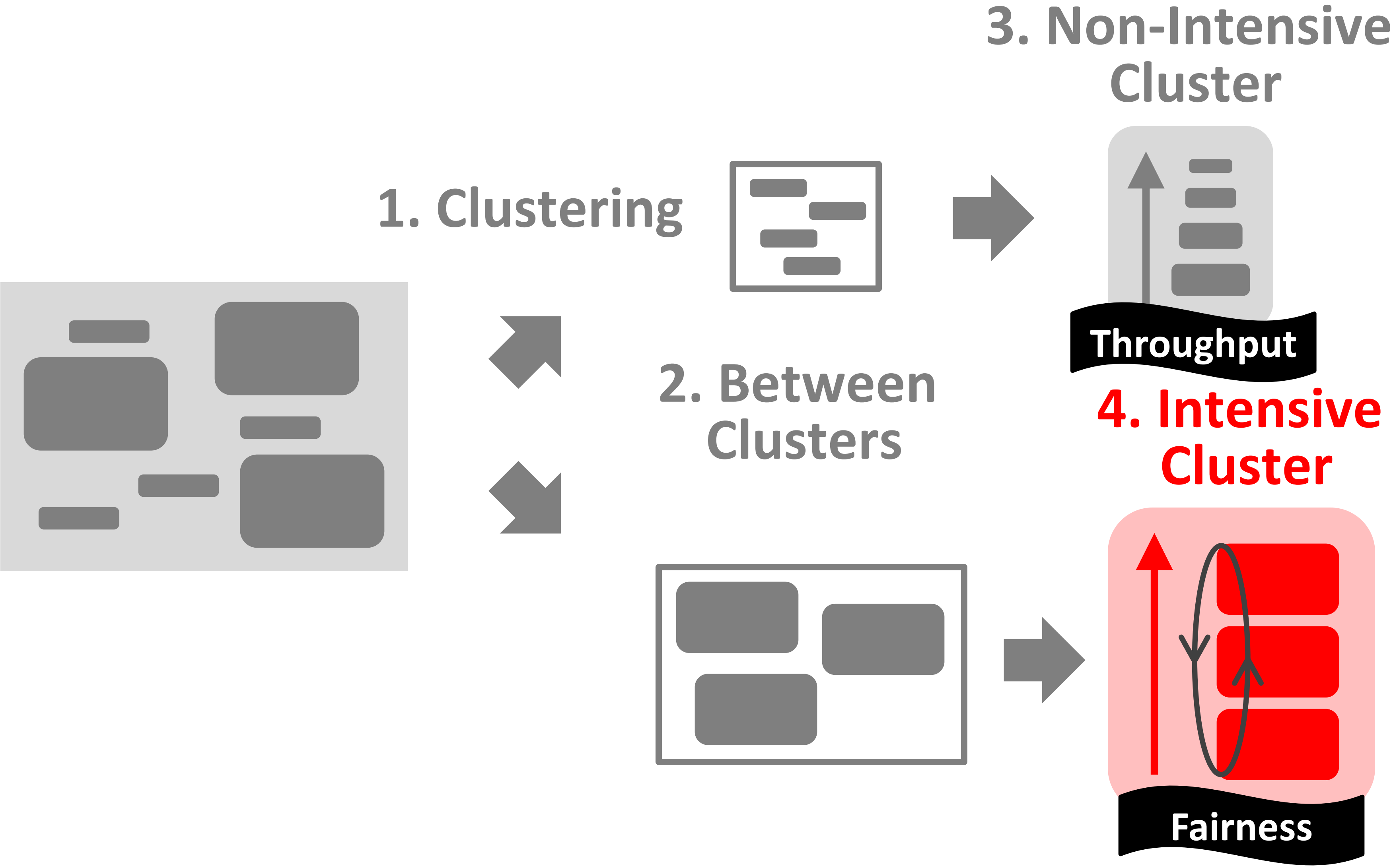
Non-Intensive Cluster

Prioritize threads according to MPKI



- **Increases system throughput**
  – Least intensive thread has the greatest potential for making progress in the processor

TCM Outline



3. Non-Intensive Cluster

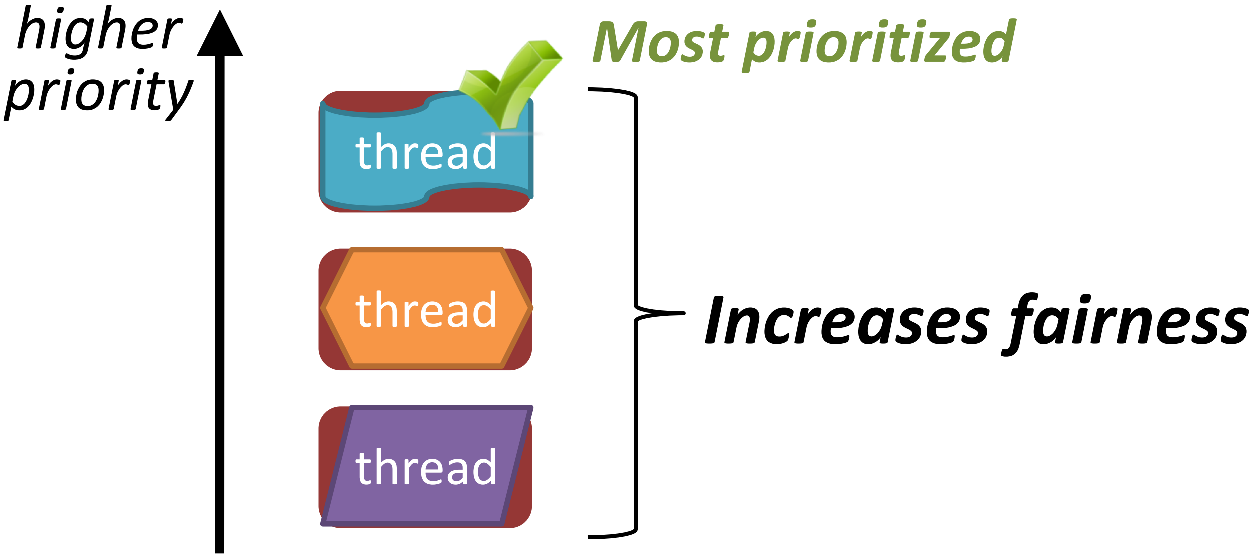
1. Clustering

2. Between Clusters

4. Intensive Cluster

Intensive Cluster

Periodically shuffle the priority of threads



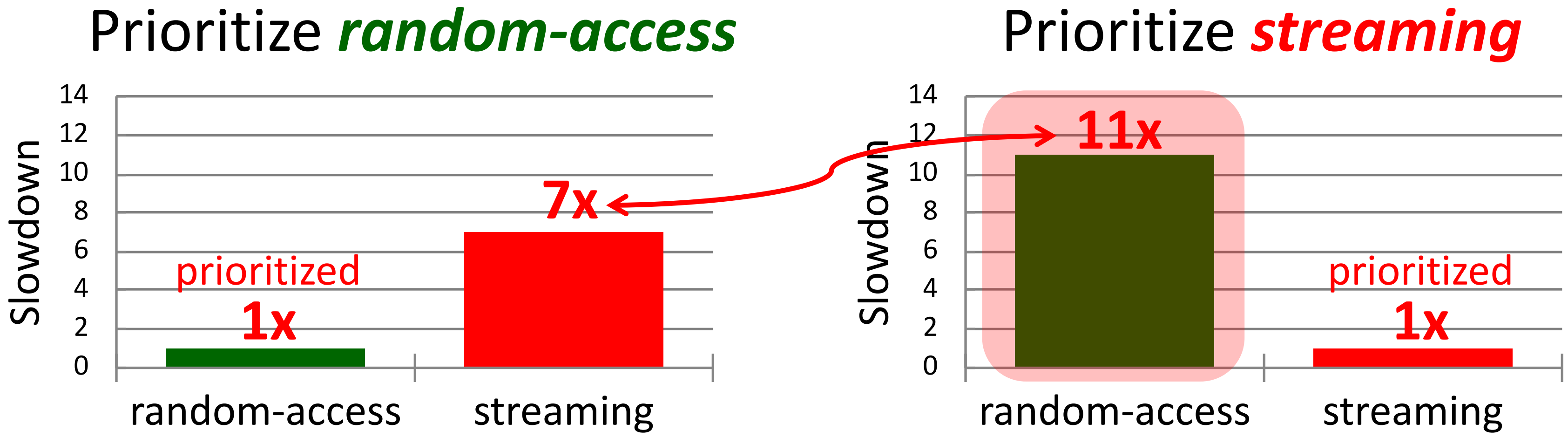
- Is treating all threads equally good enough?
- ***BUT:*** *Equal turns ≠ Same slowdown*

Case Study: A Tale of Two Threads

Case Study: Two intensive threads contending

1. ***random-access***
2. ***streaming***

Which is slowed down more easily?

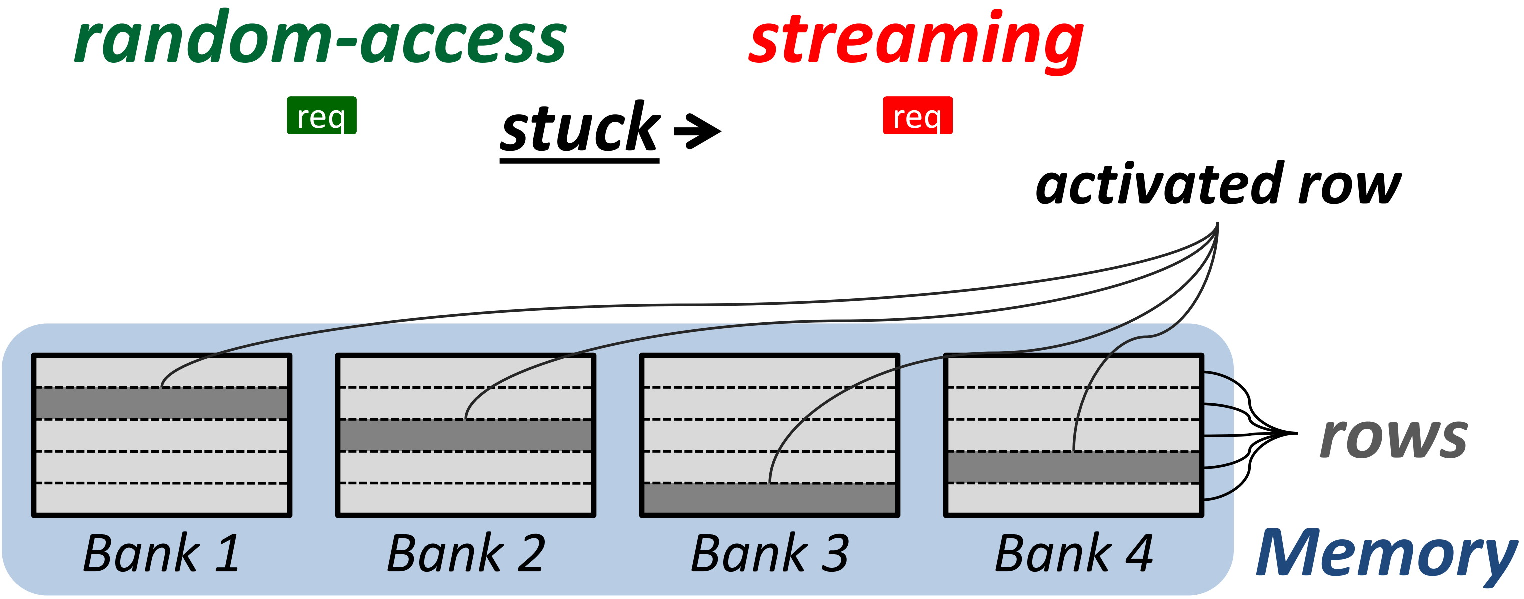


Prioritize ***random-access***

Prioritize ***streaming***

random-access thread is more easily slowed down

Why are Threads Different?

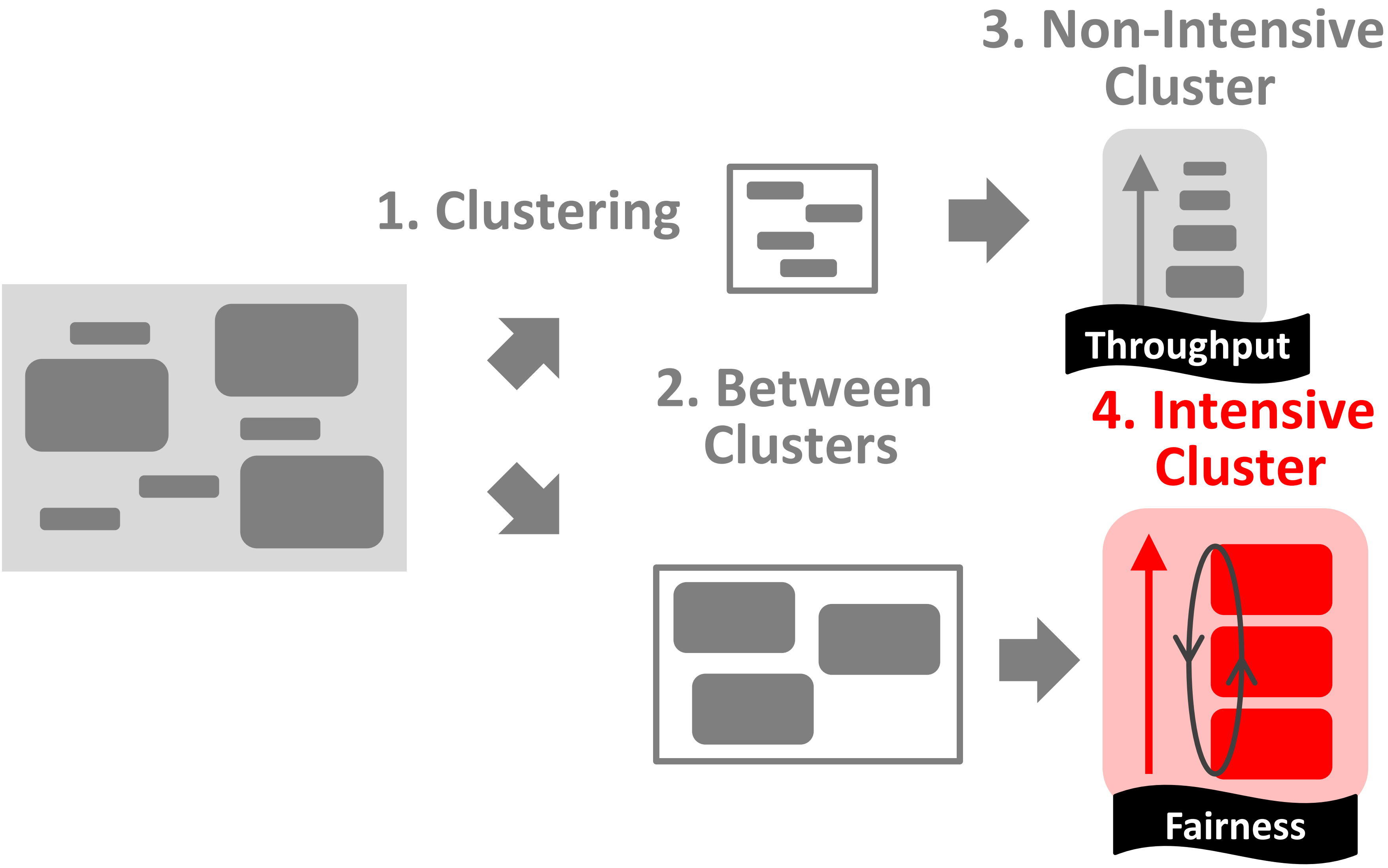


- All requests parallel
- High **bank-level parallelism**

- All requests → Same row
- High **row-buffer locality**

Vulnerable to interference

TCM Outline



1. Clustering

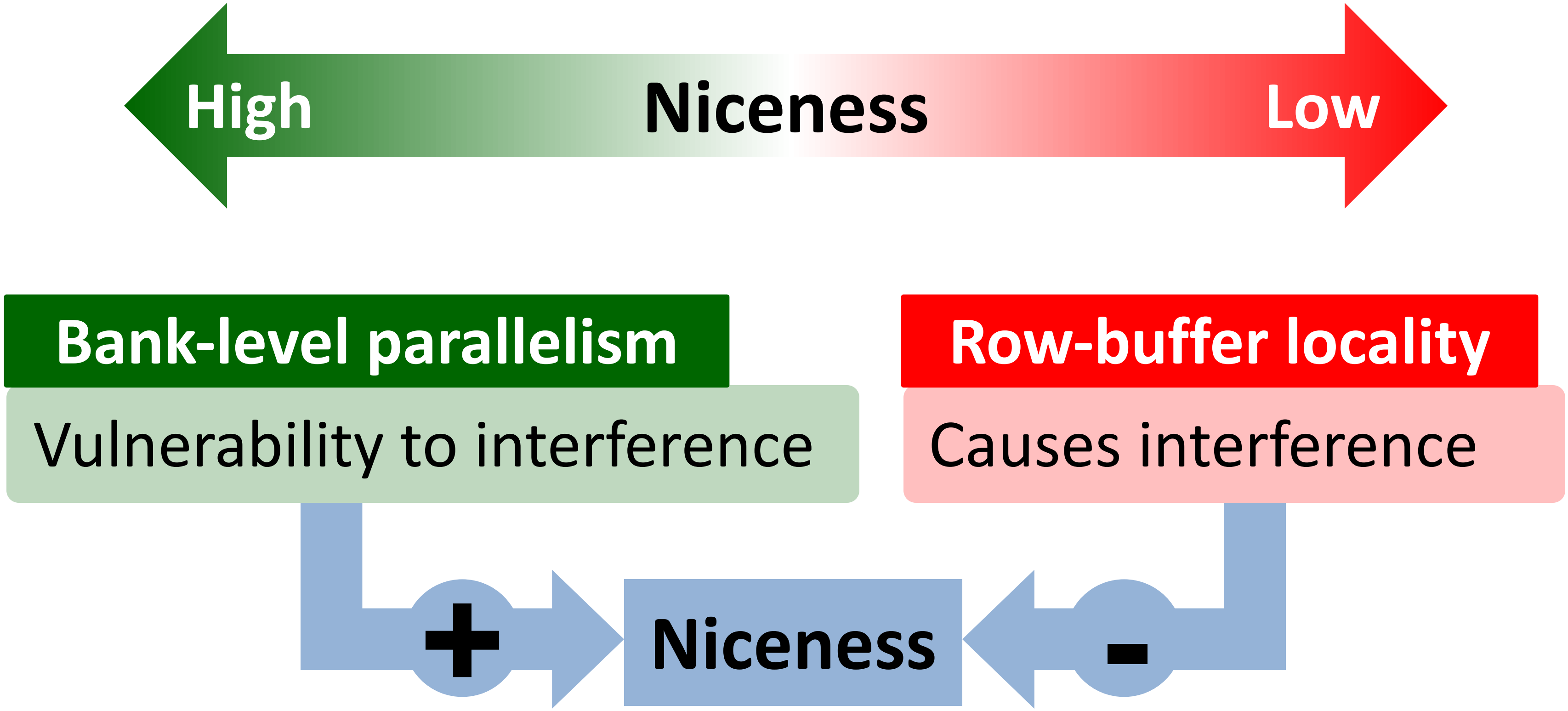
2. Between Clusters

3. Non-Intensive Cluster

4. Intensive Cluster

Niceness

How to quantify difference between threads?



High

Niceness

Low

Bank-level parallelism

Vulnerability to interference

Row-buffer locality

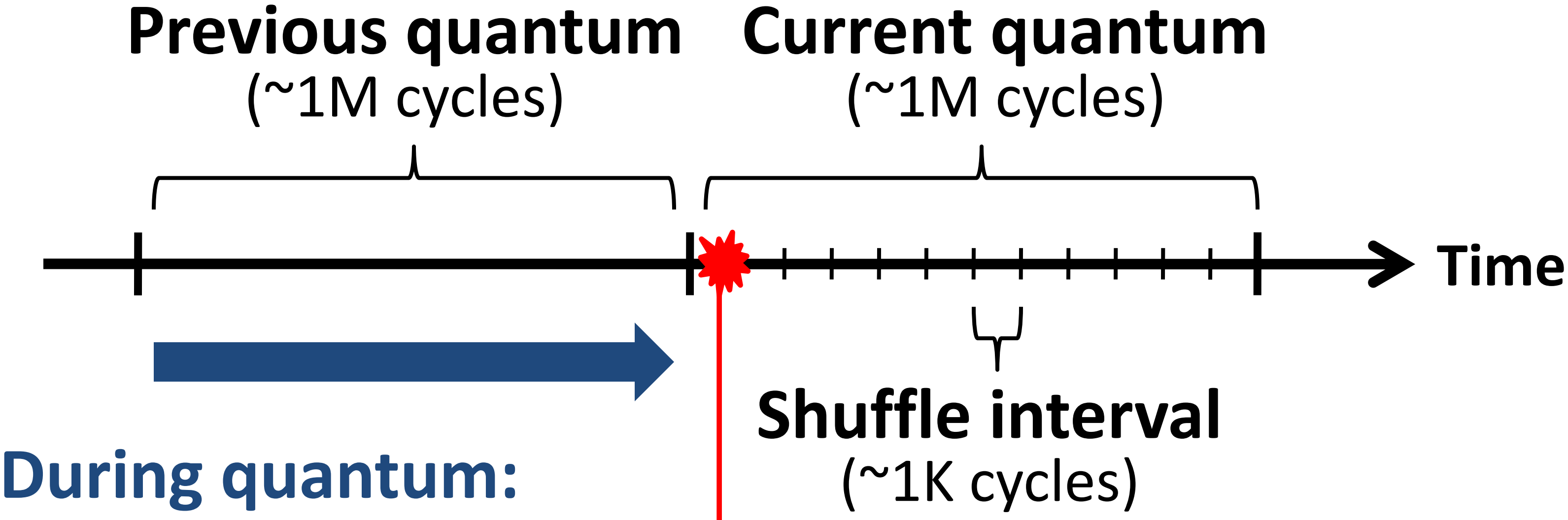
Causes interference

+

Niceness

-

TCM: Quantum-Based Operation



Previous quantum
(~1M cycles)

Current quantum
(~1M cycles)

Time

Shuffle interval
(~1K cycles)

During quantum:

- Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality

Beginning of quantum:

- Perform clustering
- Compute niceness of intensive threads

TCM: Scheduling Algorithm

1. ***Highest-rank***: Requests from higher ranked threads prioritized
   - **Non-Intensive** cluster > **Intensive** cluster
   - **Non-Intensive** cluster: lower intensity → higher rank
   - **Intensive** cluster: rank shuffling

2. ***Row-hit***: Row-buffer hit requests are prioritized

3. ***Oldest***: Older requests are prioritized

TCM: Implementation Cost

***Required storage at memory controller** (24 cores)*

Thread memory behavior	Storage
MPKI	~0.2kb
Bank-level parallelism	~0.6kb
Row-buffer locality	~2.9kb
Total	**< 4kbits**

- No computation is on the critical path

Previous Work

FRFCFS [Rixner et al., ISCA00]: Prioritizes row-buffer hits

- Thread-oblivious ➔ Low throughput & Low fairness

STFM [Mutlu et al., MICRO07]: Equalizes thread slowdowns

- Non-intensive threads not prioritized ➔ Low throughput

PAR-BS [Mutlu et al., ISCA08]: Prioritizes oldest batch of requests while preserving bank-level parallelism

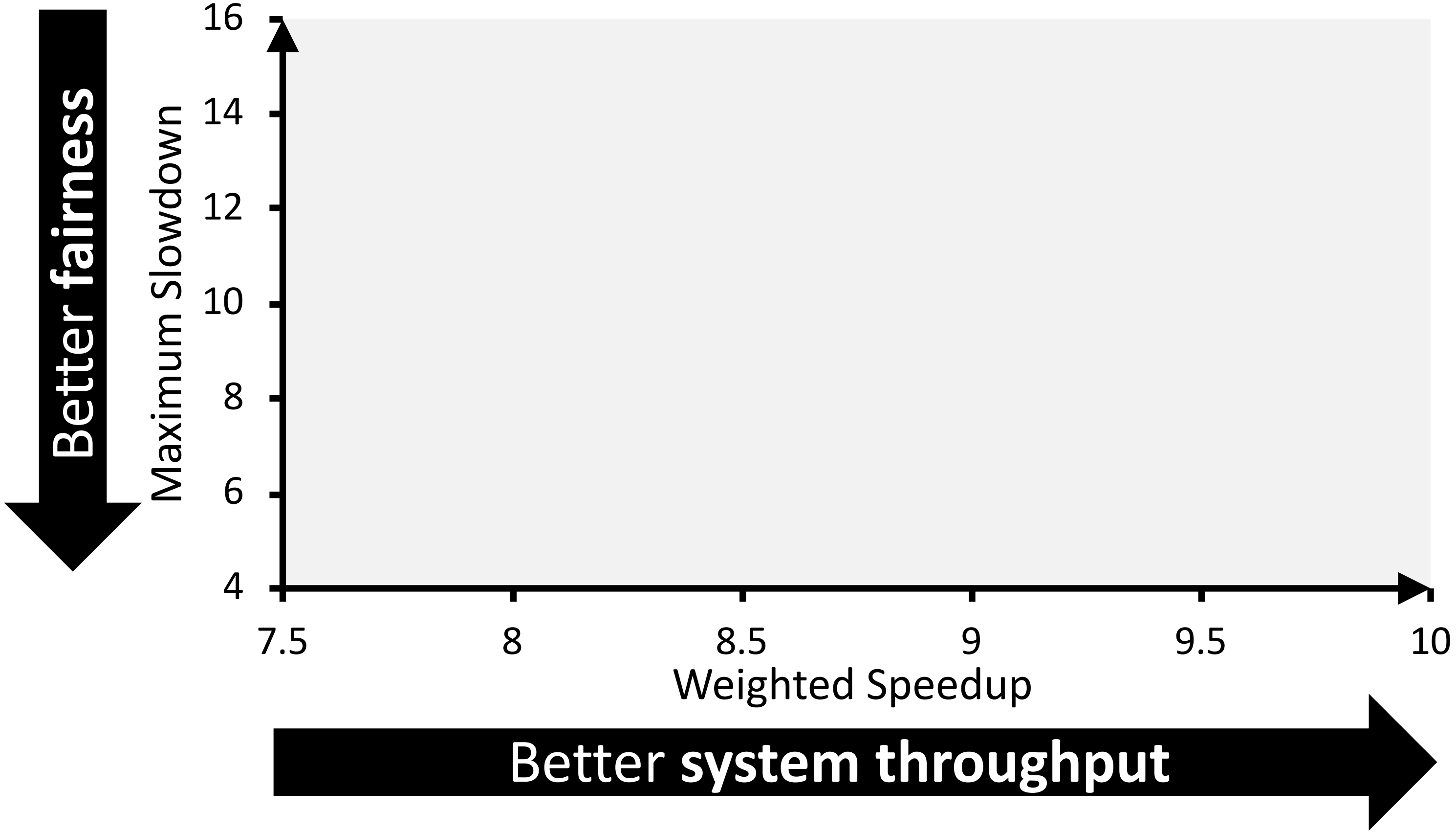
- Non-intensive threads not always prioritized ➔ Low throughput

ATLAS [Kim et al., HPCA10]: Prioritizes threads with less memory service

- Most intensive thread starves ➔ Low fairness

TCM: Throughput and Fairness

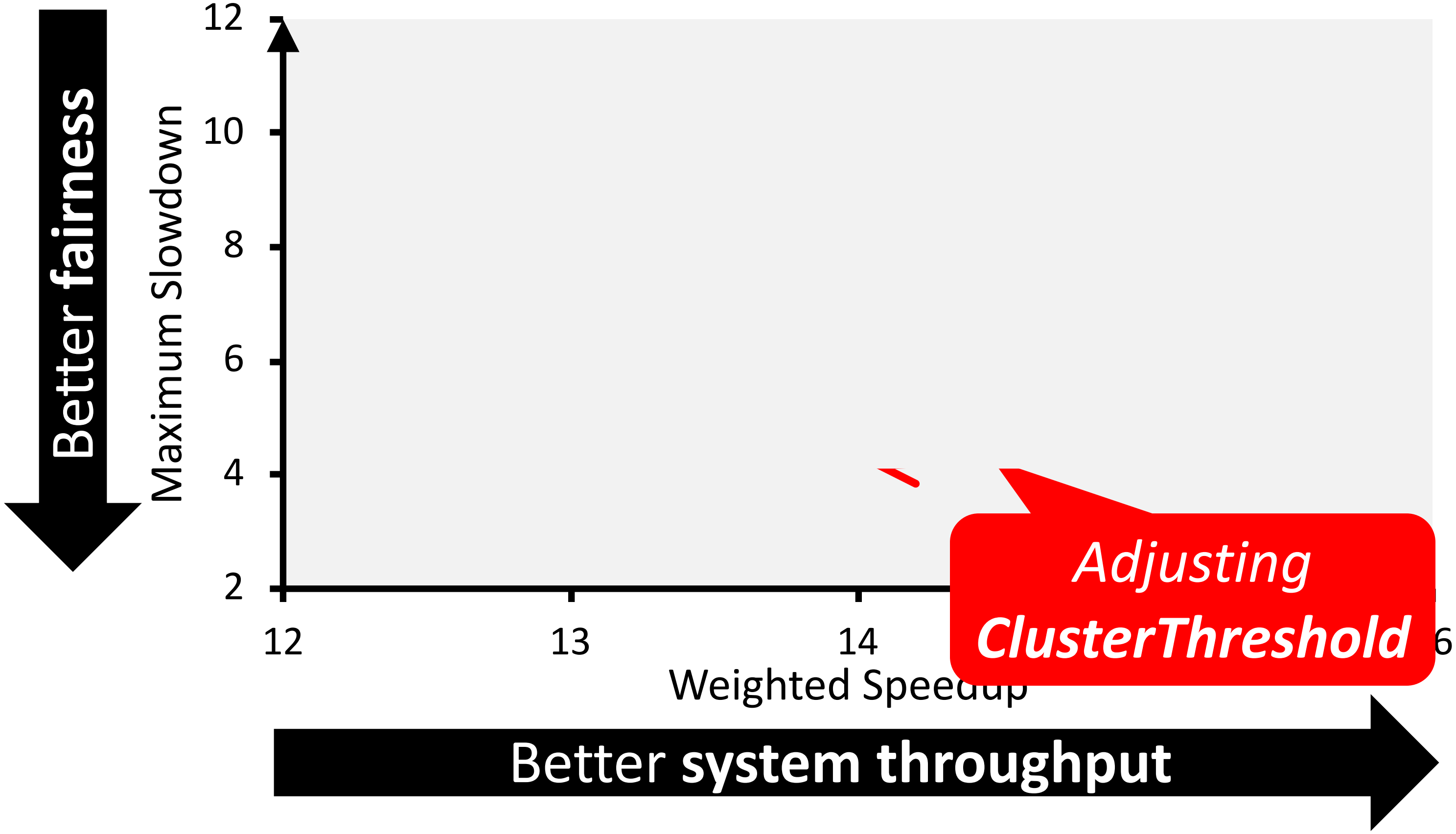
24 cores, 4 memory controllers, 96 workloads



TCM, a heterogeneous scheduling policy, provides best fairness and system throughput

TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied…



Adjusting ***ClusterThreshold***

TCM allows robust fairness-throughput tradeoff

Operating System Support

- ***ClusterThreshold*** is a tunable knob
  - OS can trade off between fairness and throughput

- Enforcing thread weights
  - OS assigns weights to threads
  - TCM enforces thread weights within each cluster

Conclusion

- No previous memory scheduling algorithm provides both high ***system throughput*** and ***fairness***
  - **Problem:** They use a single policy for all threads

- TCM groups threads into two ***clusters***
  1. Prioritize ***non-intensive*** cluster → throughput
  2. Shuffle priorities in ***intensive*** cluster → fairness
  3. Shuffling should favor ***nice*** threads → fairness

- *TCM provides the best system throughput and fairness*

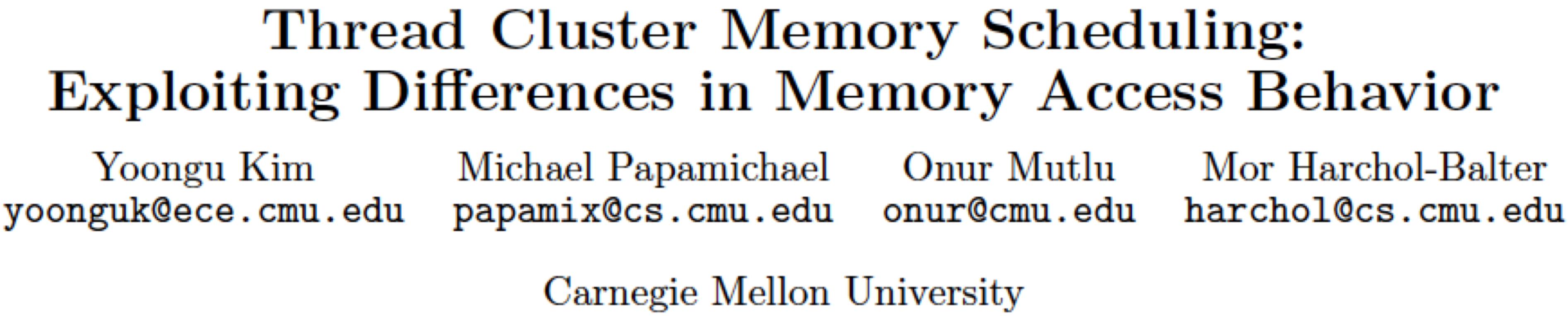
TCM Pros and Cons

- Upsides:
  - Provides both high fairness and high performance
  - Caters to the needs for different types of threads (latency vs. bandwidth sensitive)
  - (Relatively) simple

- Downsides:
  - Scalability to large buffer sizes?
  - Robustness of clustering and shuffling algorithms?
  - Ranking is still too complex?

More on TCM

- Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter, **"Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior"** *Proceedings of the 43rd International Symposium on Microarchitecture* (***MICRO***), pages 65-76, Atlanta, GA, December 2010. Slides (pptx) (pdf)



Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior

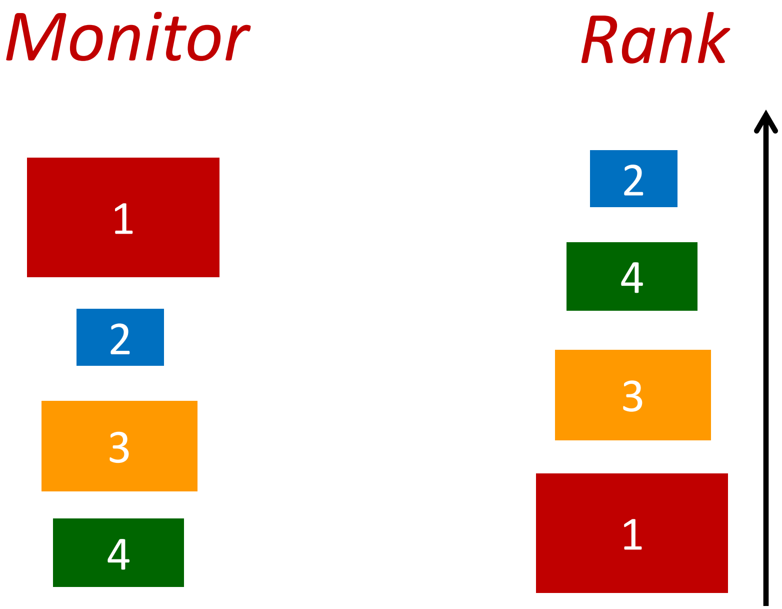
Yoongu Kim yoonguk@ece.cmu.edu
Michael Papamichael papamix@cs.cmu.edu
Onur Mutlu onur@cmu.edu
Mor Harchol-Balter harchol@cs.cmu.edu

Carnegie Mellon University

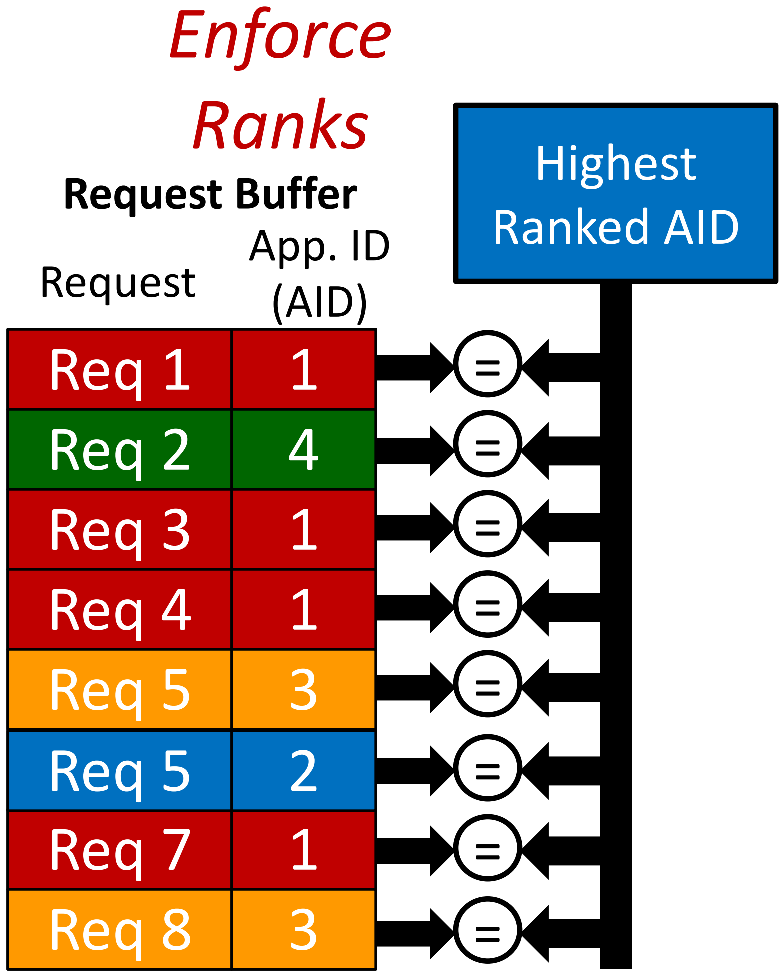
The Blacklisting Memory Scheduler

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
"The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"
Proceedings of the 32nd IEEE International Conference on Computer Design (***ICCD***),
Seoul, South Korea, October 2014. [Slides (pptx) (pdf)]

Tackling Inter-Application Interference: Application-aware Memory Scheduling



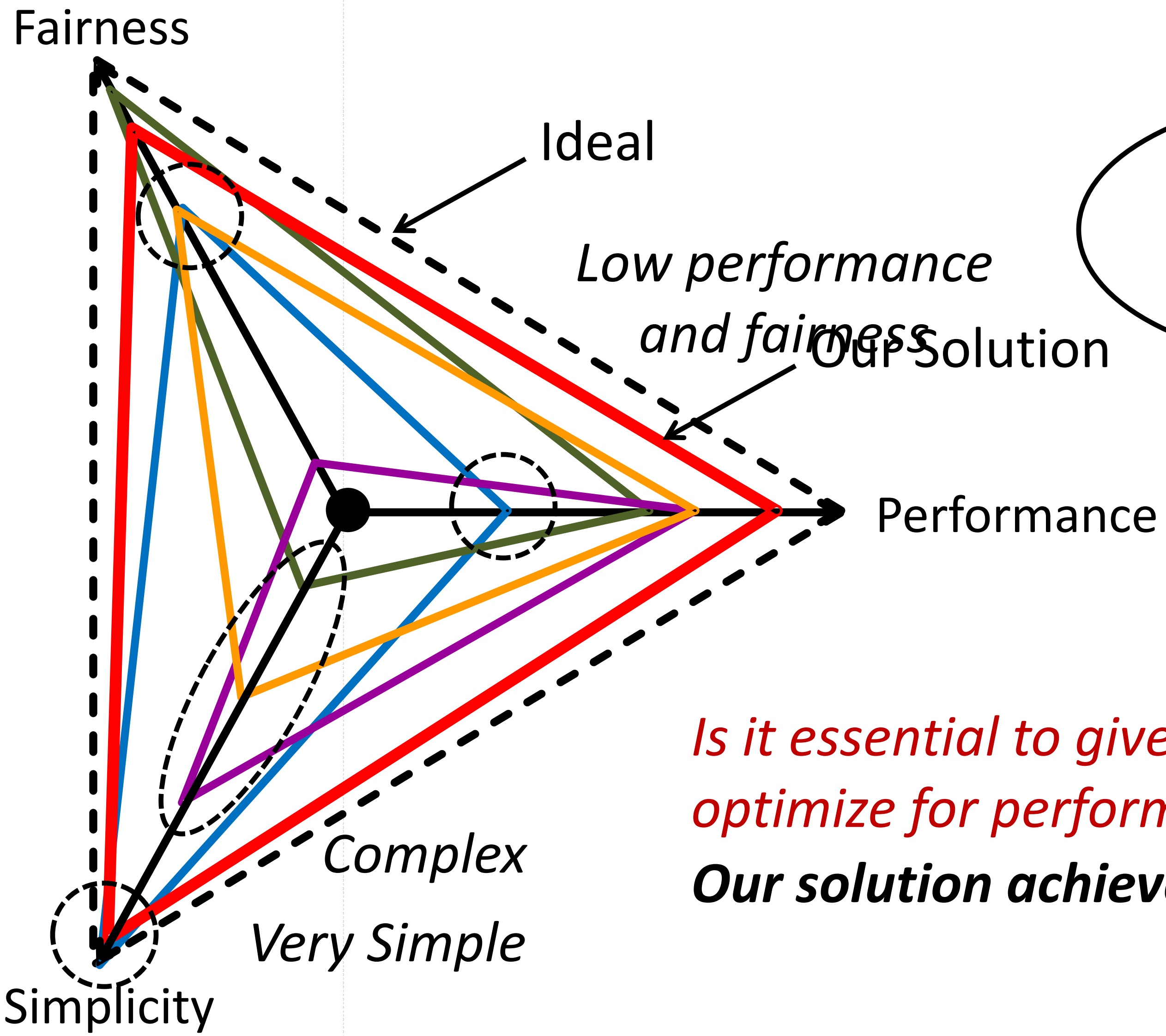
Full ranking increases critical path latency and area significantly to improve performance and fairness



Request Buffer

Request	App. ID (AID)
Req 1	1
Req 2	4
Req 3	1
Req 4	1
Req 5	3
Req 5	2
Req 7	1
Req 8	3

Performance vs. Fairness vs. Simplicity

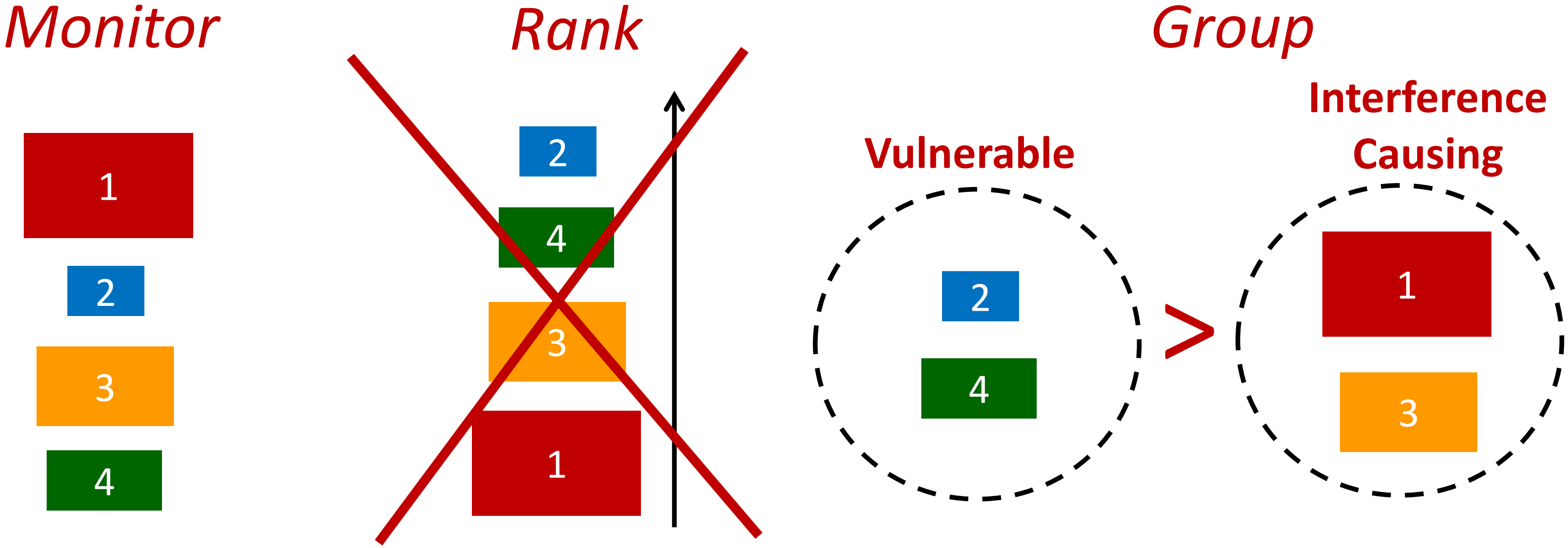


Is it essential to give up simplicity to optimize for performance and/or fairness?

Our solution achieves all three goals

Key Observation 1: Group Rather Than Rank

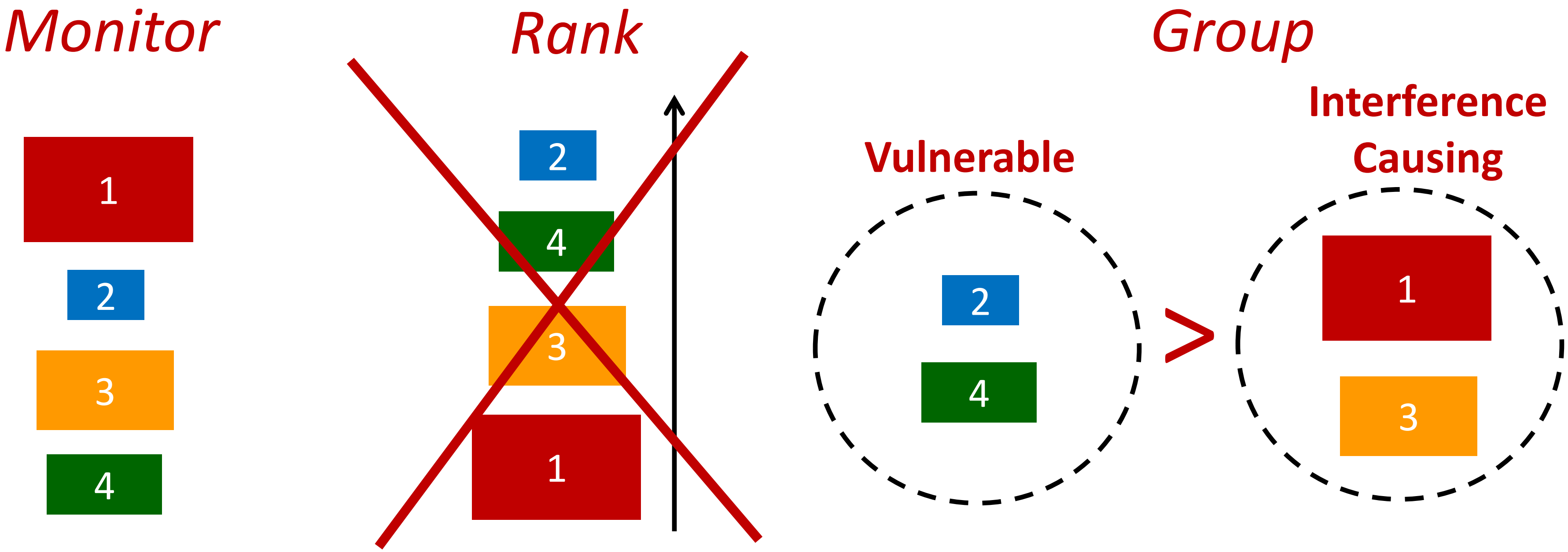
Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



Benefit 2: Lower slowdowns than ranking

Key Observation 1: Group Rather Than Rank

Observation 1: Sufficient to separate applications into two groups, rather than do full ranking



How to classify applications into groups?

Key Observation 2

Observation 2: Serving a large number of consecutive requests from an application causes interference

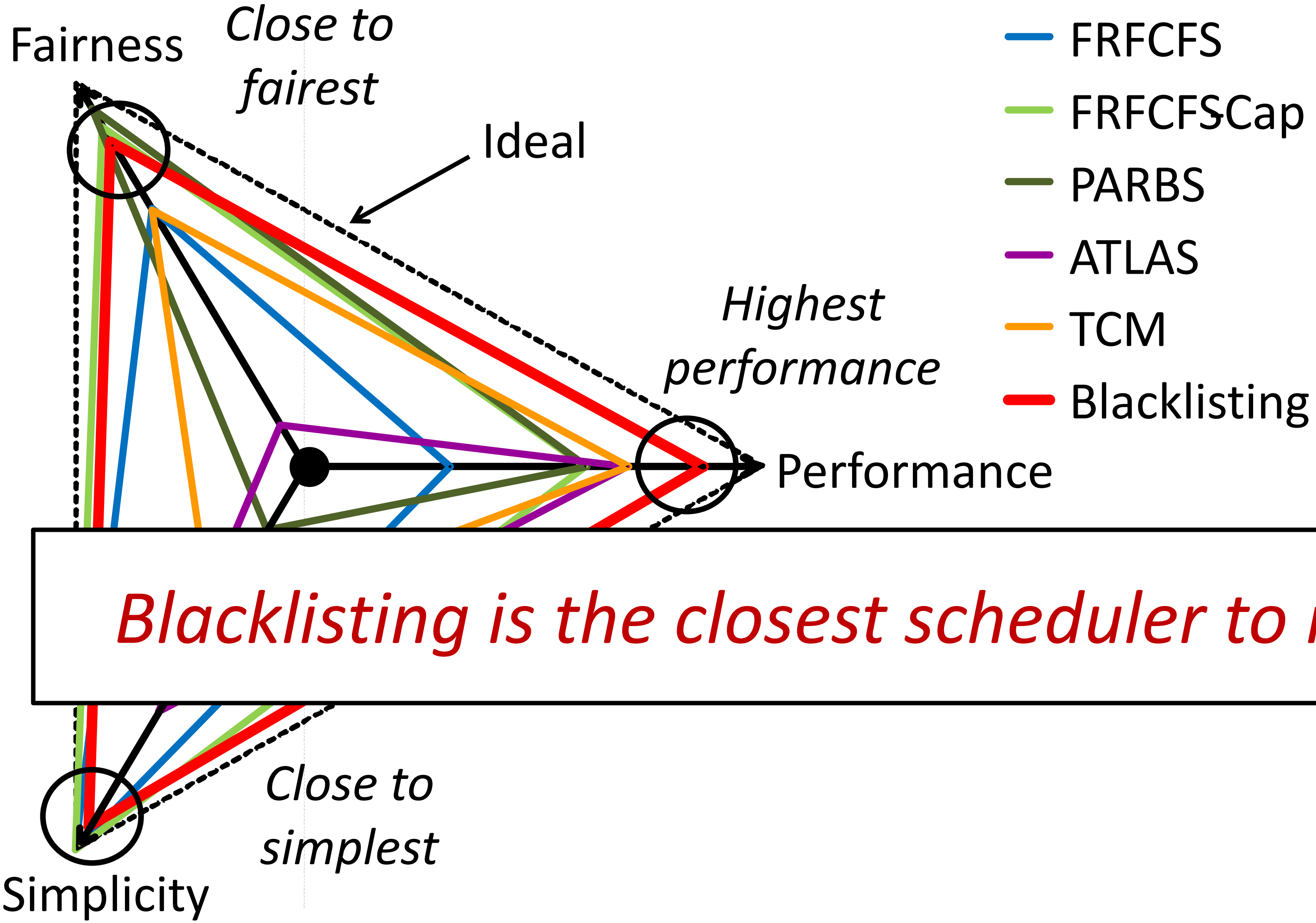
Basic Idea:

- *Group* applications with a large number of consecutive requests as *interference-causing* → *Blacklisting*
- *Deprioritize* blacklisted applications
- *Clear* blacklist periodically (1000s of cycles)

Benefits:

- *Lower complexity*
- *Finer grained grouping decisions* → *Lower unfairness*

Performance vs. Fairness vs. Simplicity



Close to fairest

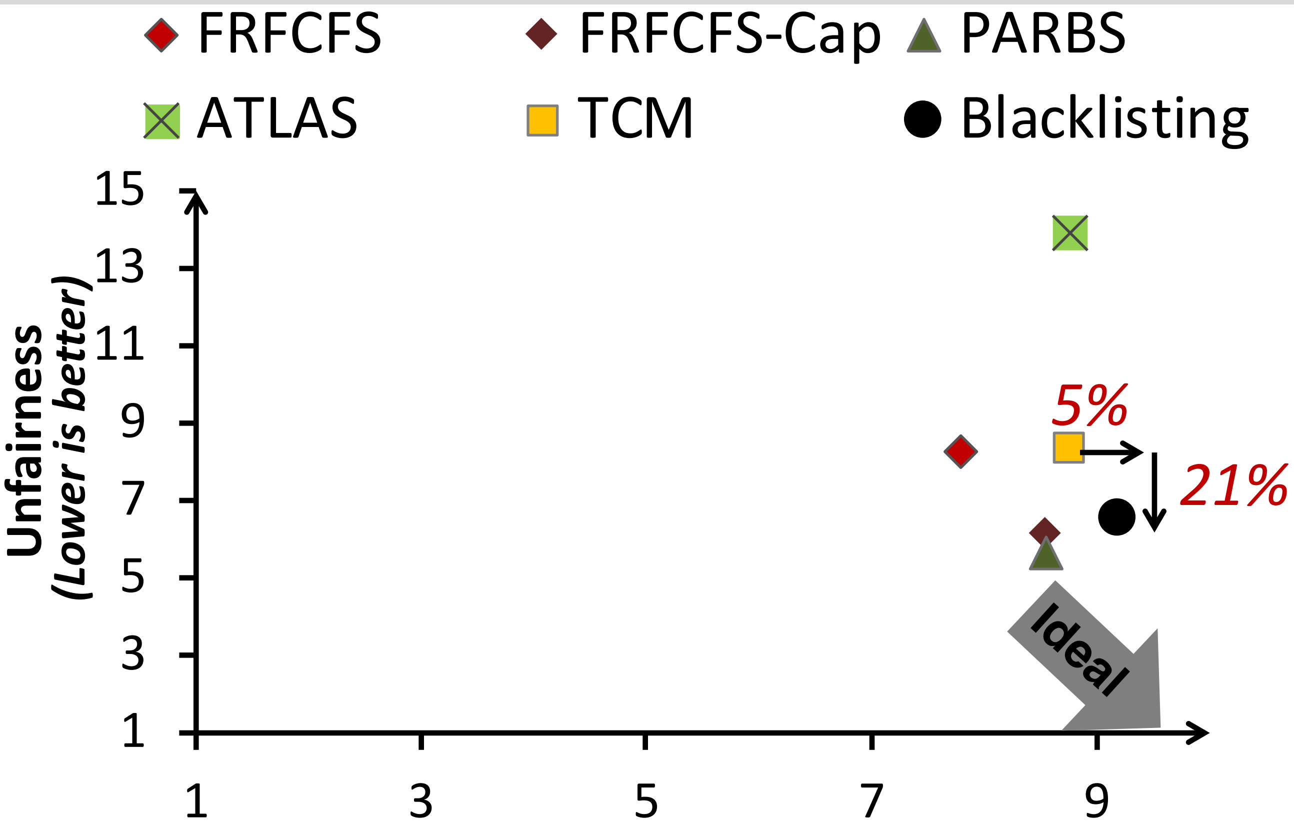
Ideal

Highest performance

Close to simplest

Blacklisting is the closest scheduler to ideal

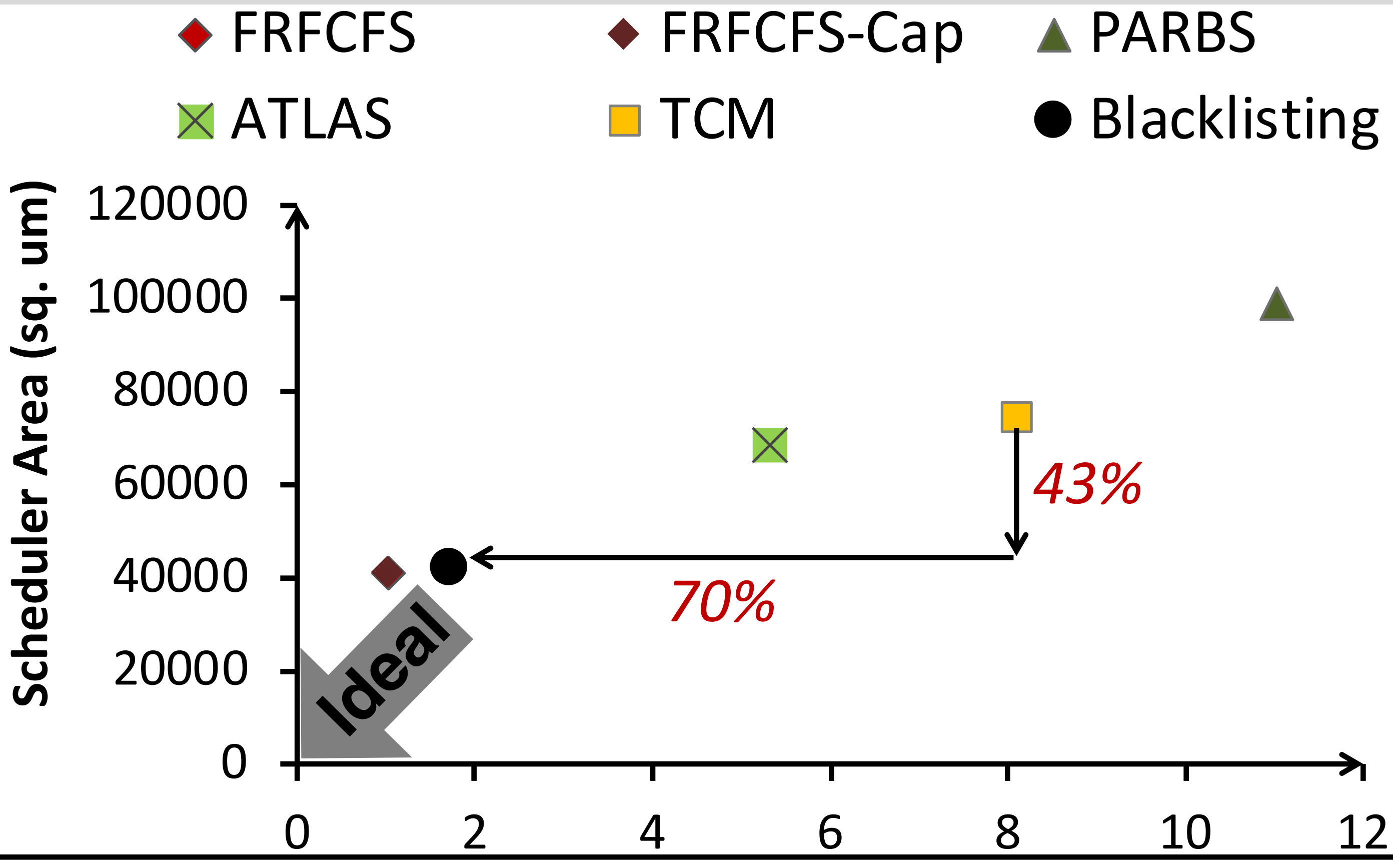
Performance and Fairness



1. Blacklisting achieves the highest performance

2. Blacklisting balances performance and fairness

Complexity



Blacklisting reduces complexity significantly

More on BLISS (I)

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
  "The Blacklisting Memory Scheduler: Achieving High Performance and Fairness at Low Cost"
  Proceedings of the 32nd IEEE International Conference on Computer Design (***ICCD***), Seoul, South Korea, October 2014.
  [Slides (pptx) (pdf)]

The Blacklisting Memory Scheduler:
Achieving High Performance and Fairness at Low Cost

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, Onur Mutlu
Carnegie Mellon University
{lsubrama,donghyu1,visesh,harshar,onur}@cmu.edu

More on BLISS: Longer Version

- Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu,
  "BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling"
  IEEE Transactions on Parallel and Distributed Systems (***TPDS***), to appear in 2016. arXiv.org version, April 2015.
  An earlier version as *SAFARI Technical Report*, TR-SAFARI-2015-004, Carnegie Mellon University, March 2015.
  [Source Code]

BLISS: Balancing Performance, Fairness and Complexity in Memory Access Scheduling

Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu

Computer Architecture

Lecture 13: Memory Interference and Quality of Service

Prof. Onur Mutlu

ETH Zürich

Fall 2021

11 November 2021

Staged Memory Scheduling

Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,
"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"
39th International Symposium on Computer Architecture (***ISCA***),
Portland, OR, June 2012.

SMS ISCA 2012 Talk

SMS: Executive Summary

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers
- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes
- **Solution:** Staged Memory Scheduling (SMS)

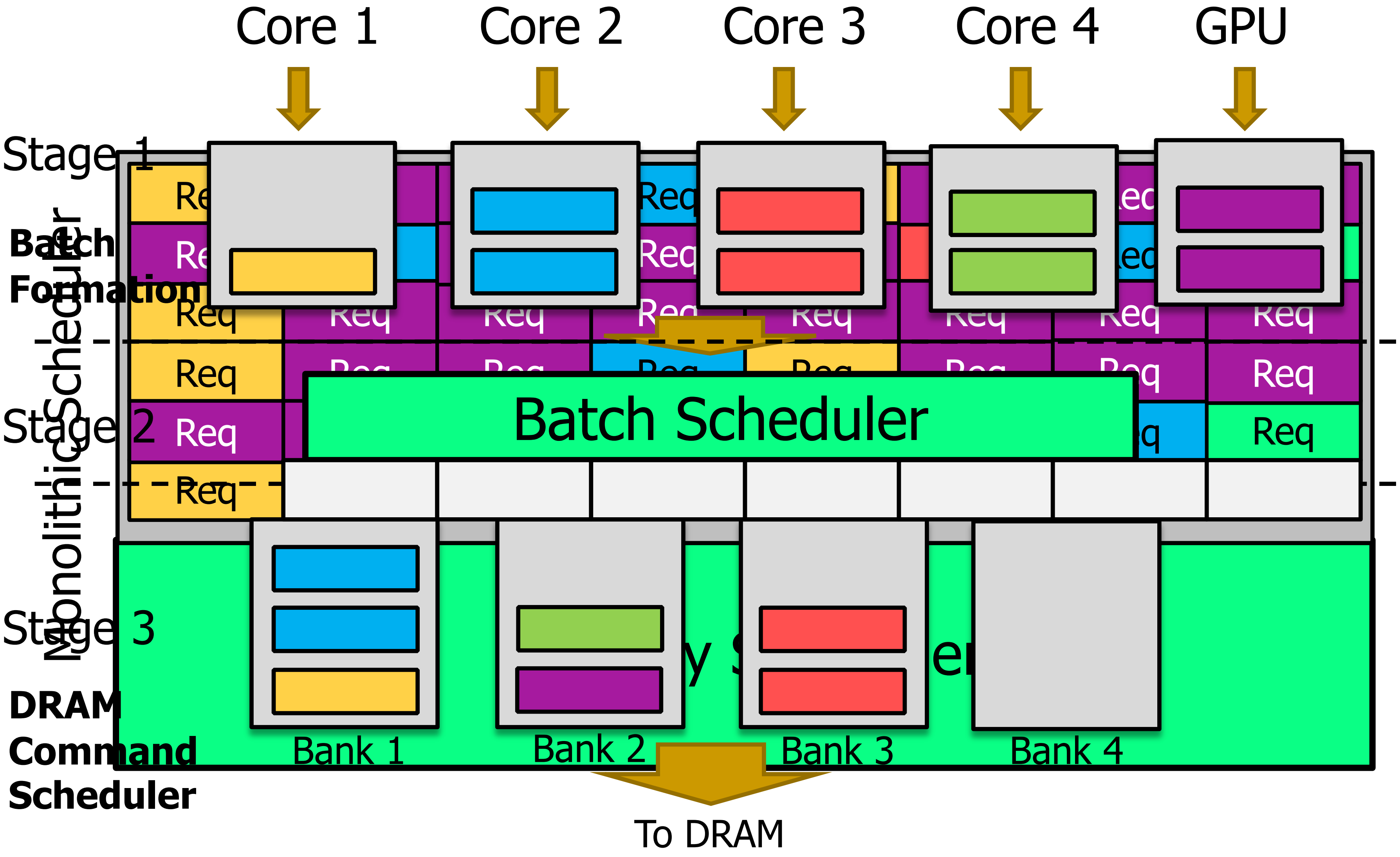
  decomposes the memory controller into three simple stages:

  1) Batch formation: maintains row buffer locality

  2) Batch scheduler: reduces interference between applications

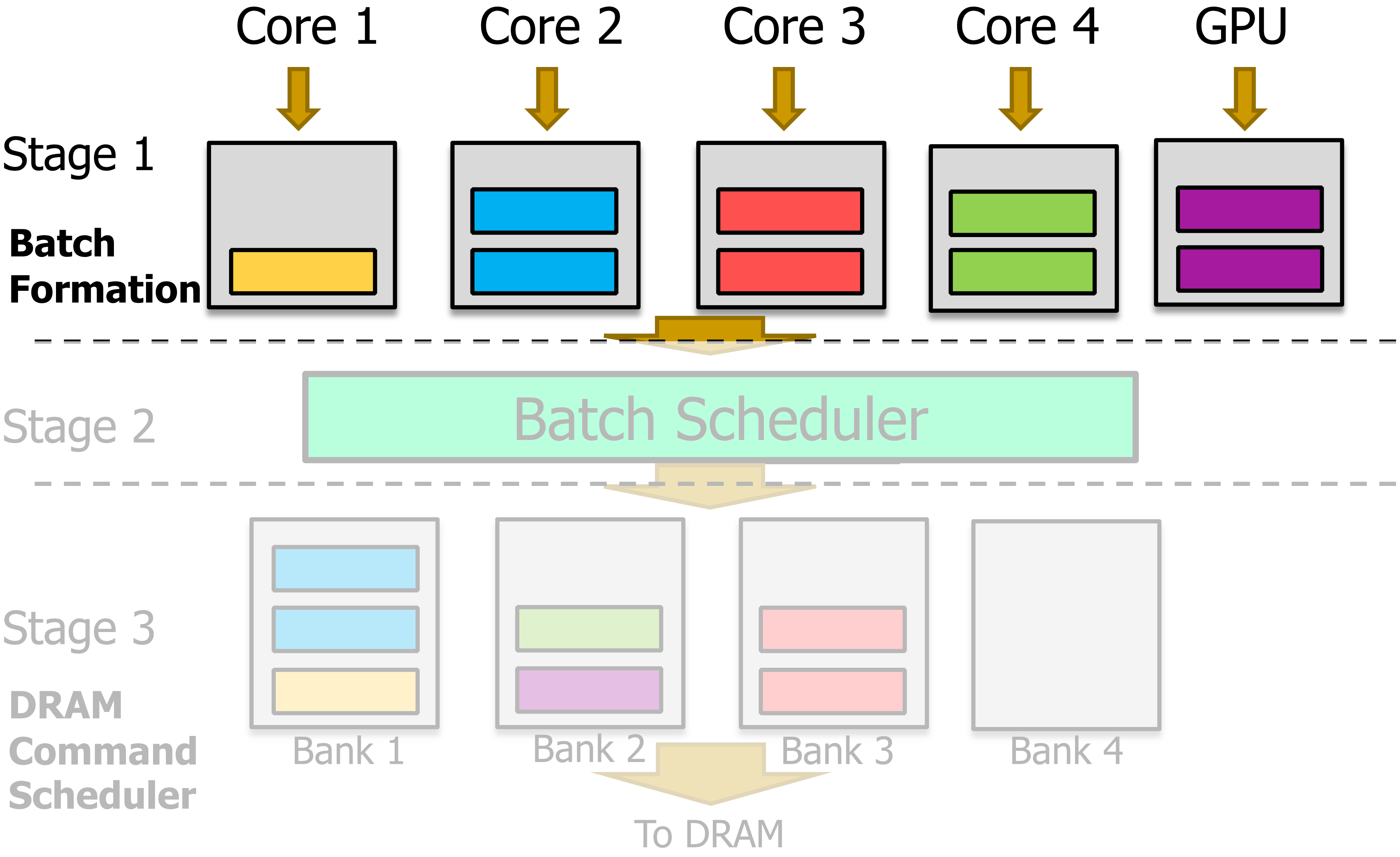
  3) DRAM command scheduler: issues requests to DRAM
- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

SMS: Staged Memory Scheduling

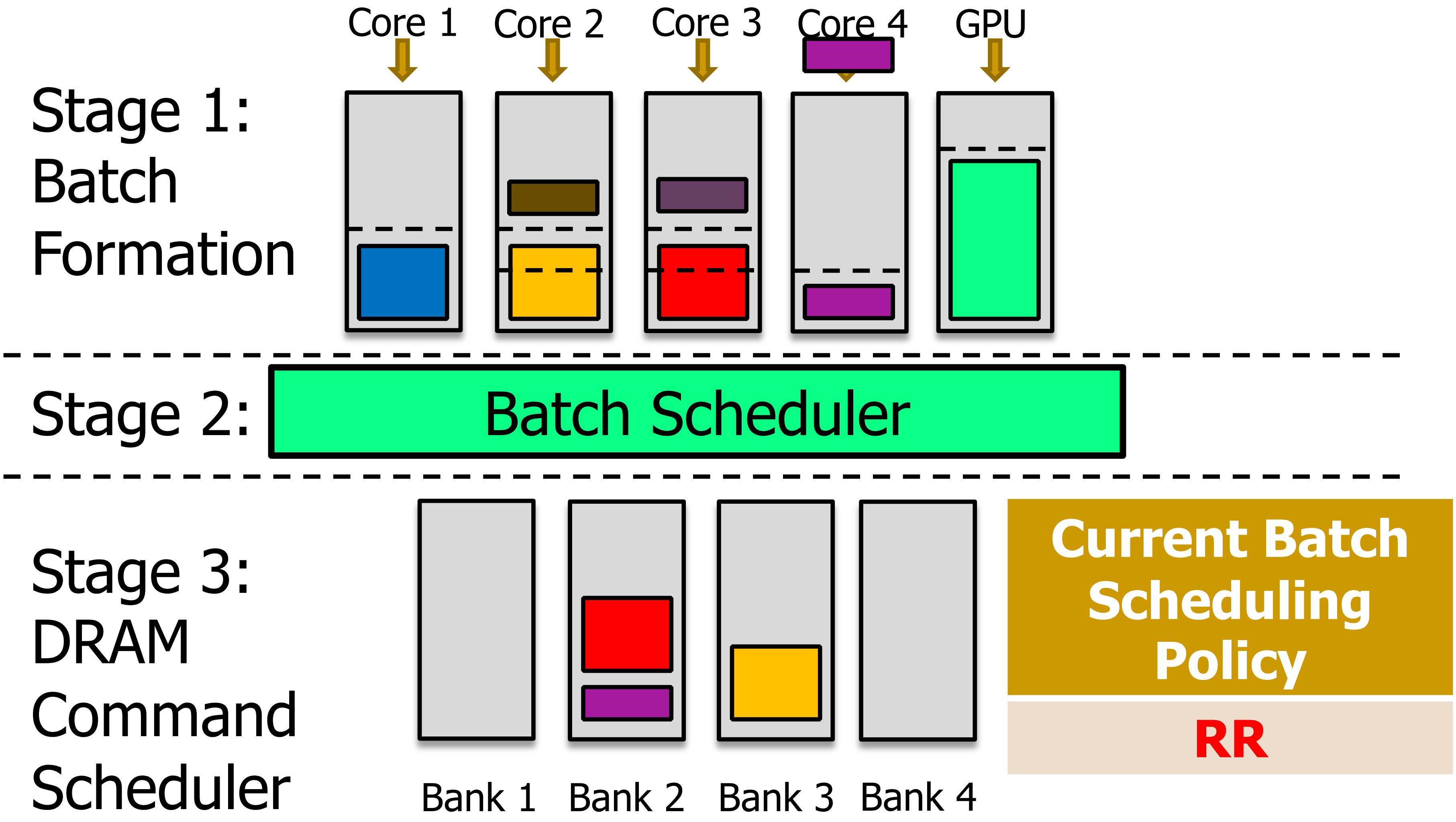


SAFARI

SMS: Staged Memory Scheduling



Putting Everything Together



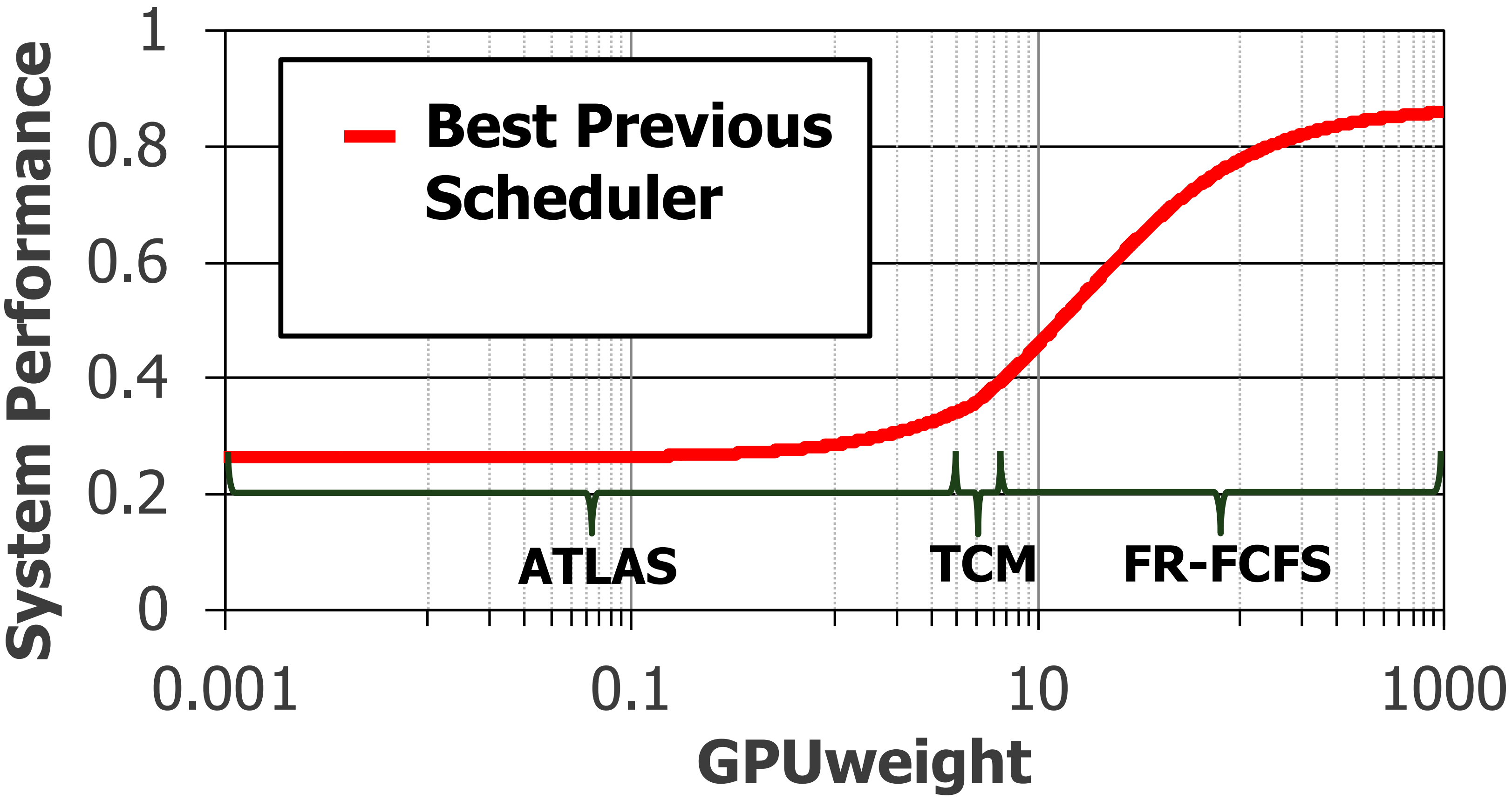
Complexity

- Compared to a row hit first scheduler, SMS consumes*
  - 66% less area
  - 46% less static power

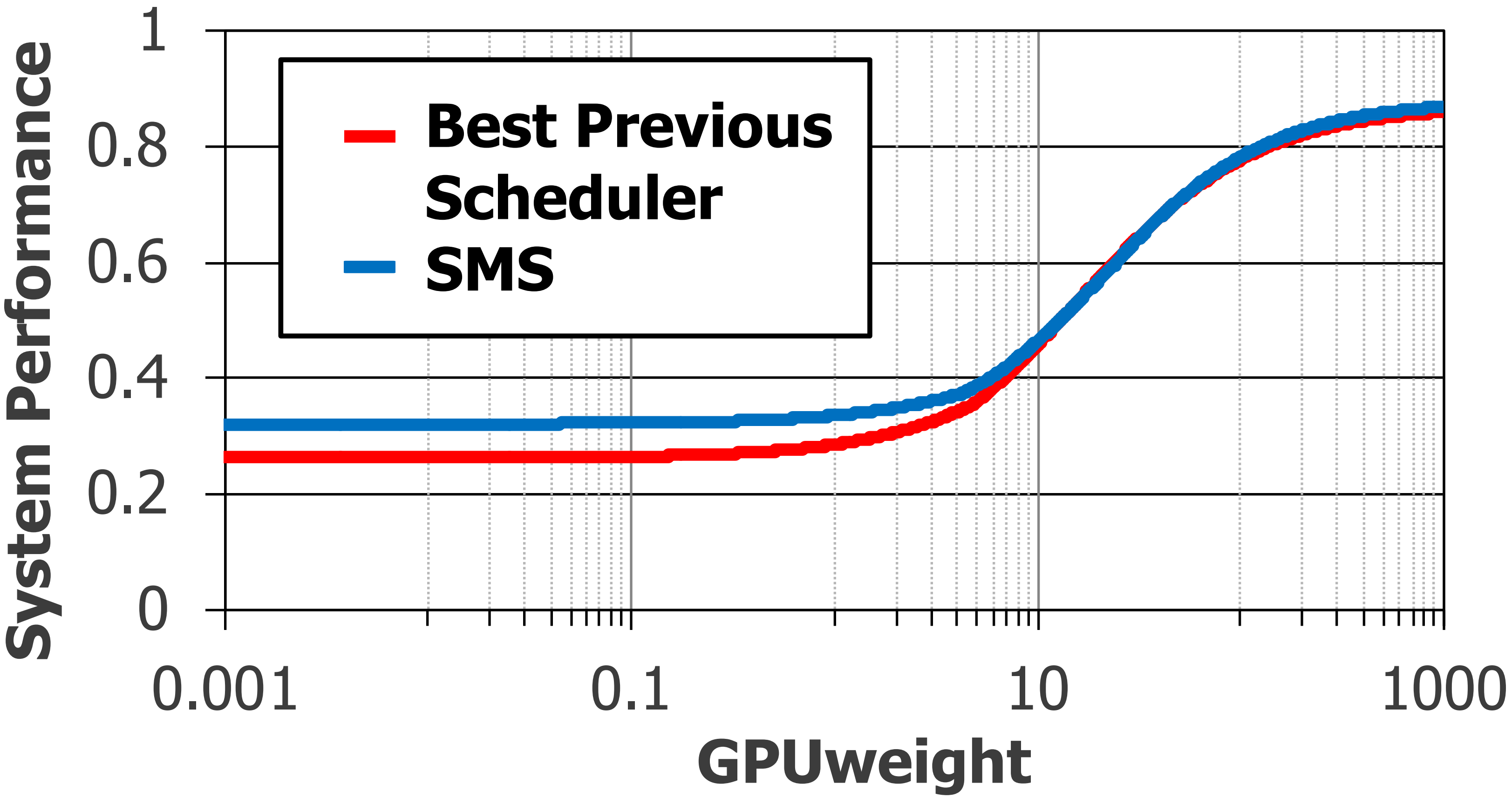
- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer size (buffering is distributed across stages)

*** Based on a Verilog model using 180nm library**

Performance at Different GPU Weights



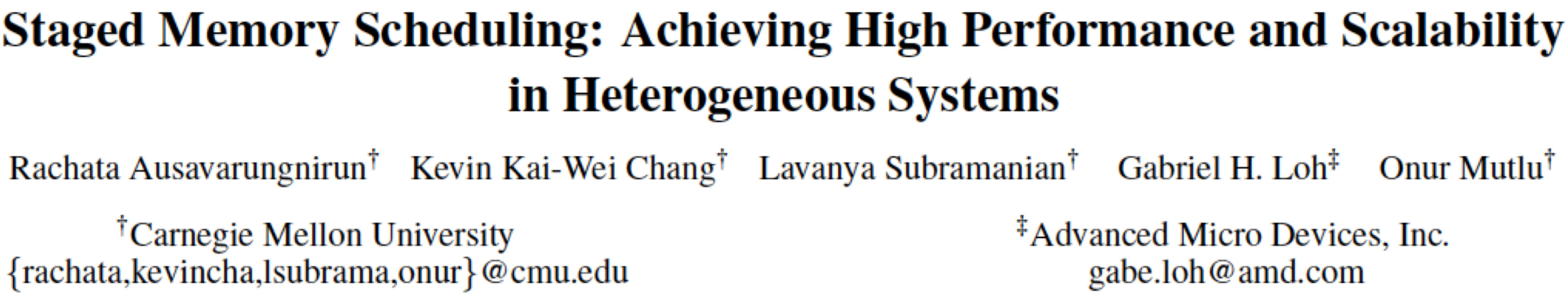
Performance at Different GPU Weights



- At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight

More on SMS

- Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu, **"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"** *Proceedings of the 39th International Symposium on Computer Architecture (**ISCA**)*, Portland, OR, June 2012. Slides (pptx)



Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems

Rachata Ausavarungnirun[†] Kevin Kai-Wei Chang[†] Lavanya Subramanian[†] Gabriel H. Loh[‡] Onur Mutlu[†]

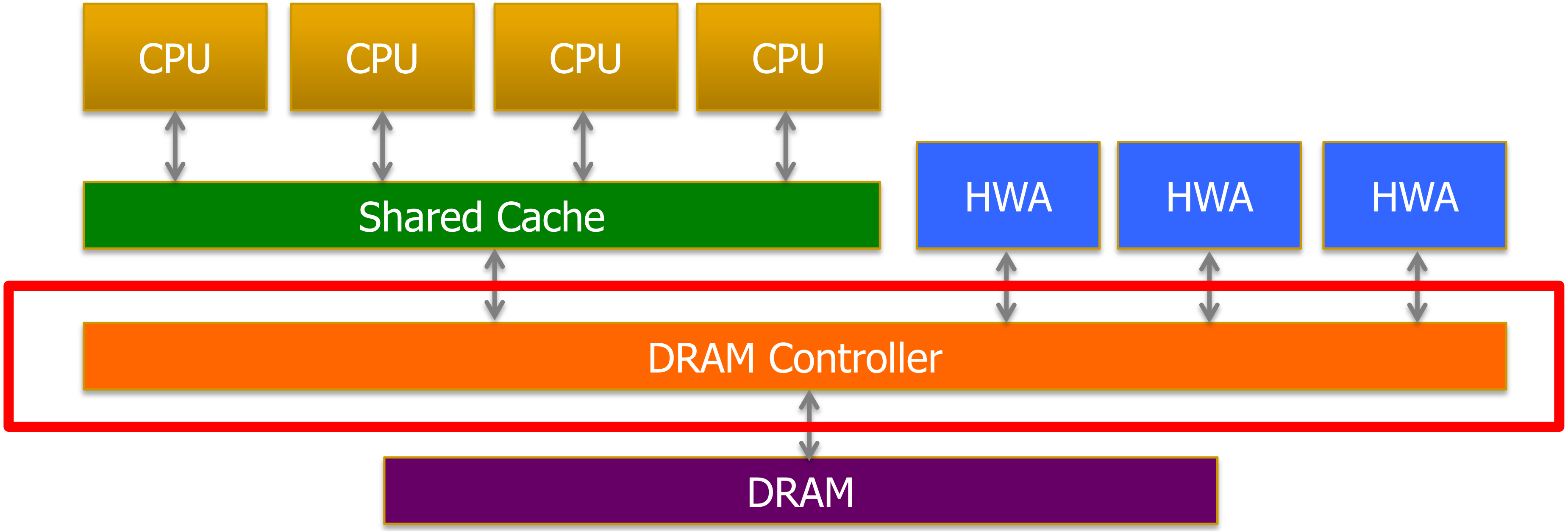
[†]Carnegie Mellon University
{rachata,kevincha,lsubrama,onur}@cmu.edu

[‡]Advanced Micro Devices, Inc.
gabe.loh@amd.com

DASH Memory Scheduler

[TACO 2016]

Current SoC Architectures



- Heterogeneous agents: CPUs and HWAs
  - HWA : Hardware Accelerator
- Main memory is shared by CPUs and HWAs → Interference

How to schedule memory requests from CPUs and HWAs
to mitigate interference?

DASH Scheduler: Executive Summary

- Problem: Hardware accelerators (HWAs) and CPUs share the same memory subsystem and interfere with each other in main memory
- Goal: Design a memory scheduler that improves CPU performance while meeting HWAs' deadlines
- Challenge: Different HWAs have different memory access characteristics and different deadlines, which current schedulers do not smoothly handle
  - Memory-intensive and long-deadline HWAs significantly degrade CPU performance *when they become high priority* (due to slow progress)
  - Short-deadline HWAs sometimes miss their deadlines *despite high priority*
- Solution: DASH Memory Scheduler
  - Prioritize HWAs over CPU anytime when the HWA is not making good progress
  - Application-aware scheduling for CPUs and HWAs
- Key Results:

  1) Improves CPU performance for a wide variety of workloads by 9.5%

  2) Meets 100% deadline met ratio for HWAs
- DASH source code freely available on our GitHub

Goal of Our Scheduler (DASH)

- Goal: Design a memory scheduler that
  - Meets GPU/accelerators' frame rates/deadlines *and*
  - Achieves high CPU performance

- Basic Idea:
  - *Different CPU applications and hardware accelerators have different memory requirements*
  - Track progress of different agents and prioritize accordingly

Key Observation:
Distribute Priority for Accelerators

- GPU/accelerators need priority to meet deadlines
- Worst case prioritization not always the best
- Prioritize when they are **not** on track to meet a deadline

Distributing priority over time mitigates impact of accelerators on CPU cores' requests

Key Observation:
Not All Accelerators are Equal

- **Long-deadline** accelerators are more likely to **meet** their deadlines
- **Short-deadline** accelerators are more likely to **miss** their deadlines

Schedule short-deadline accelerators based on worst-case memory access time

Key Observation: Not All CPU cores are Equal

- **Memory-intensive** cores are much **less vulnerable** to interference
- **Memory non-intensive** cores are much **more vulnerable** to interference

Prioritize accelerators over memory-intensive cores to ensure accelerators do not become urgent

DASH Summary: Key Ideas and Results

- *Distribute priority for HWAs*
- *Prioritize HWAs over memory-intensive CPU cores even when not urgent*
- *Prioritize short-deadline-period HWAs based on worst case estimates*

Improves CPU performance by 7-21%

Meets (almost) 100% of deadlines for HWAs

DASH: Scheduling Policy

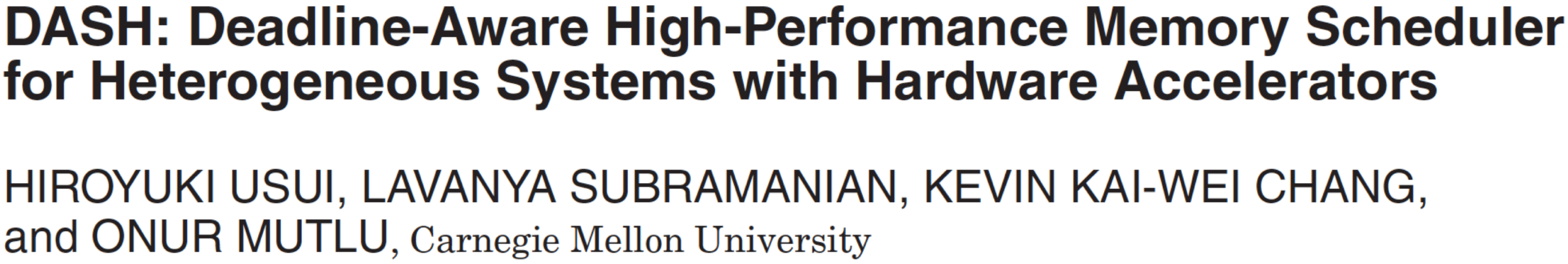
- DASH scheduling policy

  1. Short-deadline-period HWAs with high priority
  2. Long-deadline-period HWAs with high priority
  3. Memory non-intensive CPU applications
  4. Long-deadline-period HWAs with low priority
  5. Memory-intensive CPU applications
  6. Short-deadline-period HWAs with low priority

Switch probabilistically

More on DASH

- Hiroyuki Usui, Lavanya Subramanian, Kevin Kai-Wei Chang, and Onur Mutlu,
"DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators"
ACM Transactions on Architecture and Code Optimization (***TACO***), Vol. 12, January 2016.
Presented at the 11th HiPEAC Conference, Prague, Czech Republic, January 2016.
[Slides (pptx) (pdf)]
[Source Code]

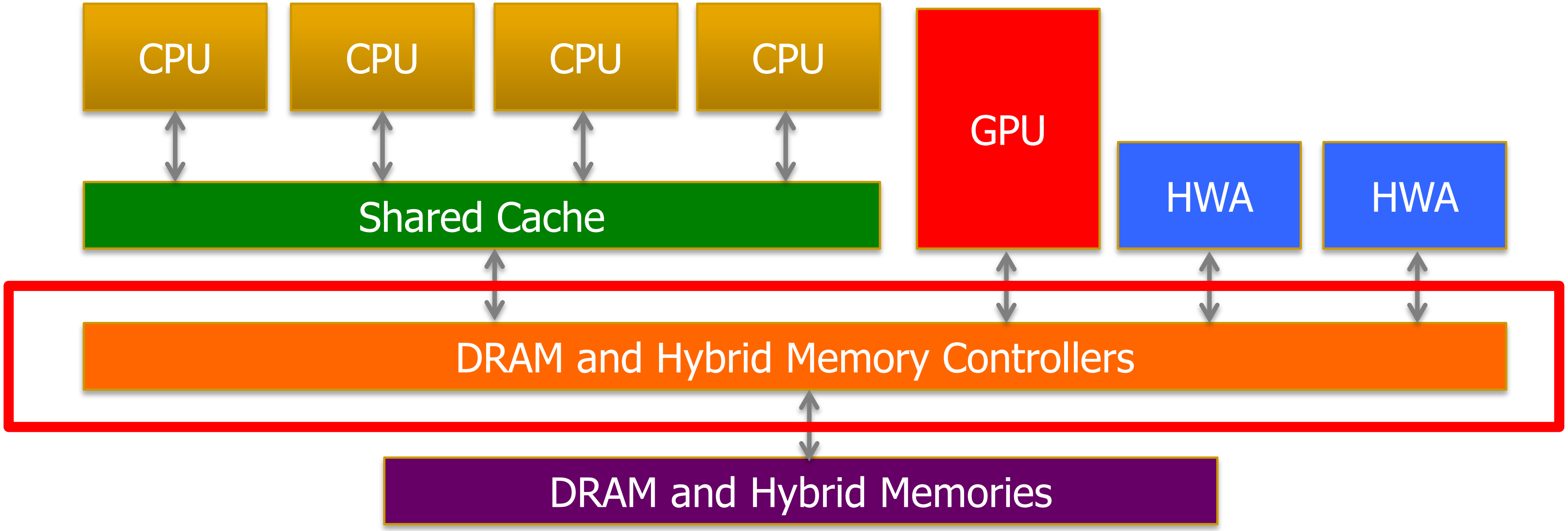


DASH: Deadline-Aware High-Performance Memory Scheduler for Heterogeneous Systems with Hardware Accelerators

HIROYUKI USUI, LAVANYA SUBRAMANIAN, KEVIN KAI-WEI CHANG, and ONUR MUTLU, Carnegie Mellon University

Predictable Performance: Strong Memory Service Guarantees

Goal: Predictable Performance in Complex Systems



- Heterogeneous agents: CPUs, GPUs, and HWAs
- Main memory interference between CPUs, GPUs, HWAs

How to allocate resources to heterogeneous agents to mitigate interference and provide predictable performance?

Strong Memory Service Guarantees

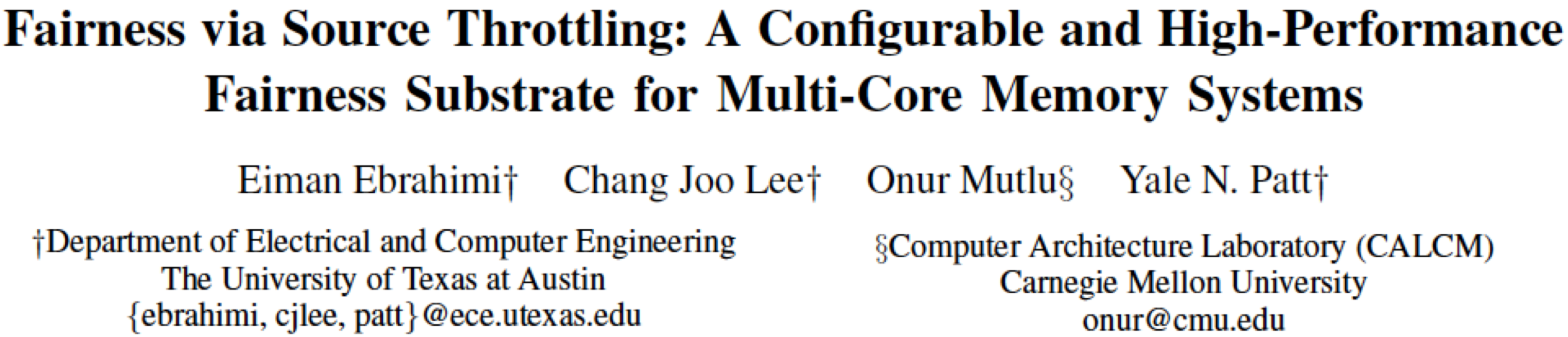
- Goal: Satisfy performance/SLA requirements in the presence of shared main memory, heterogeneous agents, and hybrid memory/storage

- Approach:
  - Develop techniques/models to accurately estimate the performance loss of an application/agent in the presence of resource sharing
  - Develop mechanisms (hardware and software) to enable the resource partitioning/prioritization needed to achieve the required performance levels for all applications
  - All the while providing high system performance

- Subramanian et al., "MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems," HPCA 2013.
- Subramanian et al., "The Application Slowdown Model," MICRO 2015.

Predictable Performance Readings (I)

- Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems"** *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, pages 335-346, Pittsburgh, PA, March 2010. Slides (pdf)



Fairness via Source Throttling: A Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems

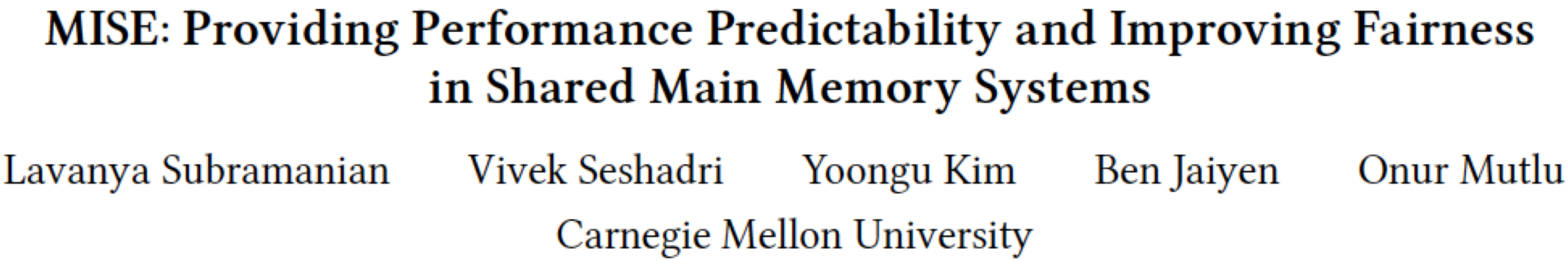
Eiman Ebrahimi† Chang Joo Lee† Onur Mutlu§ Yale N. Patt†

†Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, cjlee, patt}@ece.utexas.edu

§Computer Architecture Laboratory (CALCM)
Carnegie Mellon University
onur@cmu.edu

Predictable Performance Readings (II)

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,
"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"
*Proceedings of the 19th International Symposium on High-Performance Computer Architecture (**HPCA**)*, Shenzhen, China, February 2013. Slides (pptx)

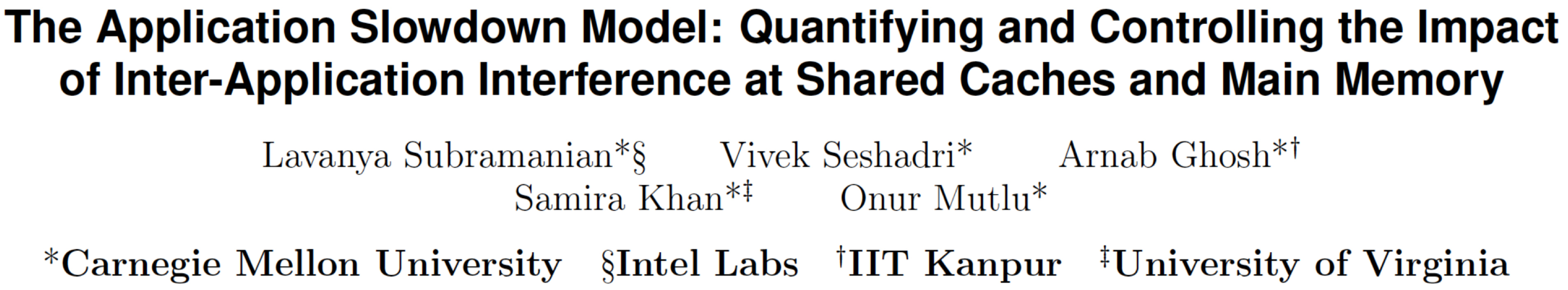


MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu
Carnegie Mellon University

Predictable Performance Readings (III)

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  *Proceedings of the 48th International Symposium on Microarchitecture (**MICRO**)*, Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]



The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian[*§] Vivek Seshadri[*] Arnab Ghosh[*†]
Samira Khan[*‡] Onur Mutlu[*]

[*]Carnegie Mellon University [§]Intel Labs [†]IIT Kanpur [‡]University of Virginia

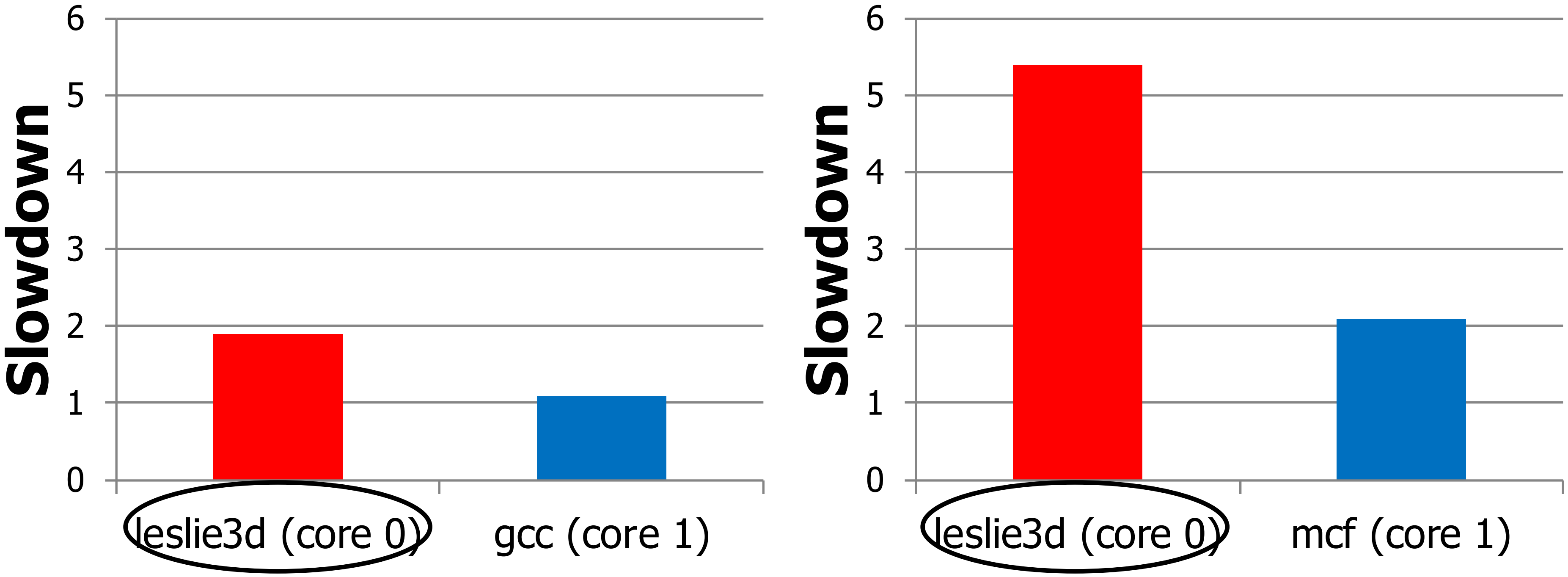
MISE:
Providing Performance Predictability in Shared Main Memory Systems

Lavanya Subramanian, Vivek Seshadri,
Yoongu Kim, Ben Jaiyen, Onur Mutlu

SAFARI

Carnegie Mellon

Unpredictable Application Slowdowns



An application's performance depends on which application it is running with

Need for Predictable Performance

- There is a need for predictable performance
  - When multiple applications share resources
  - Especially if some applications require performance guarantees

Our Goal: Predictable performance
in the presence of memory interference

- Example 2: In server systems
  - Different users' jobs consolidated onto the same server
  - Need to provide bounded slowdowns to critical jobs

Outline

1. Estimate Slowdown

2. Control Slowdown

Outline

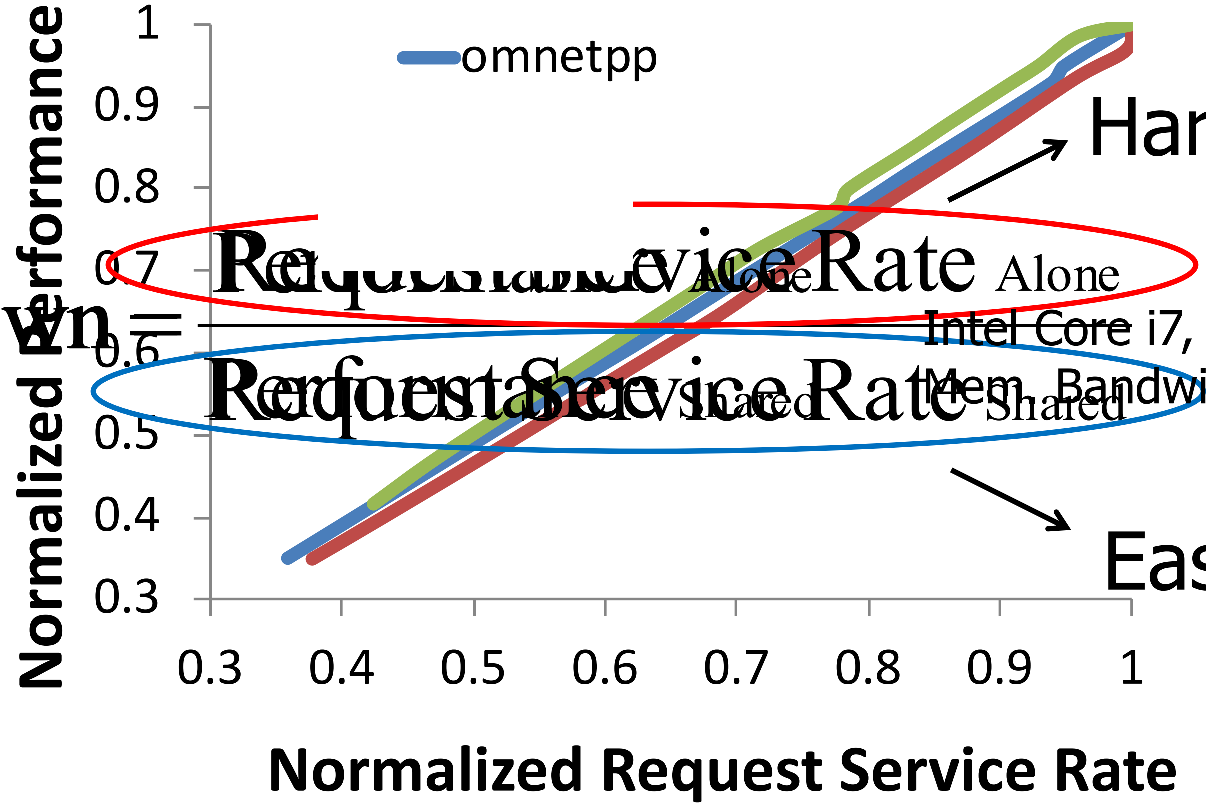
1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Slowdown: Definition

$$\text{Slowdown} = \frac{\text{Performance}_{\text{Alone}}}{\text{Performance}_{\text{Shared}}}$$

Key Observation 1

For a memory bound application,
Performance ∝ Memory request service rate



$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}}{\text{Request Service Rate}_{\text{Shared}}}$$

Harder

Easy

Intel Core i7, 4 cores
Mem. Bandwidth: 8.5 GB/s

Key Observation 2

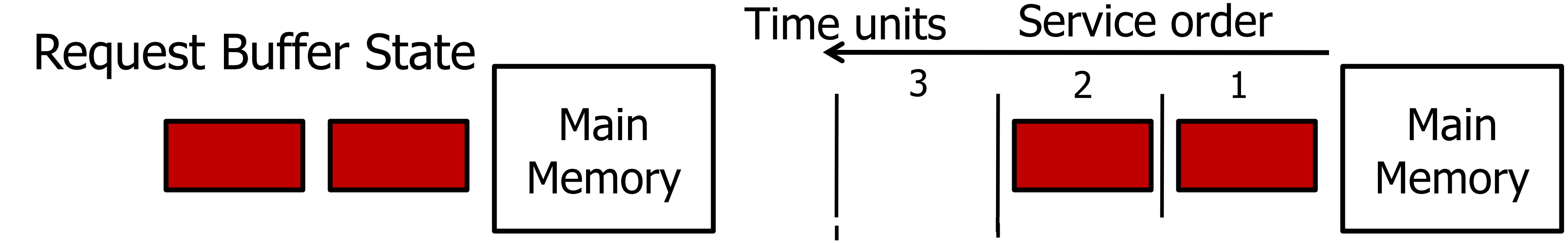
Request Service Rate $_{\text{Alone}}$ ($\text{RSR}_{\text{Alone}}$) of an application can be estimated by giving the application highest priority in accessing memory

Highest priority → Little interference

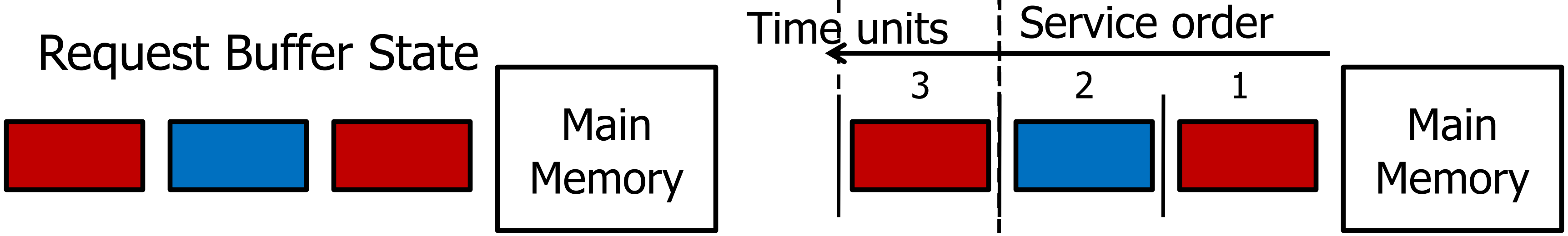
(almost as if the application were run alone)

Key Observation 2

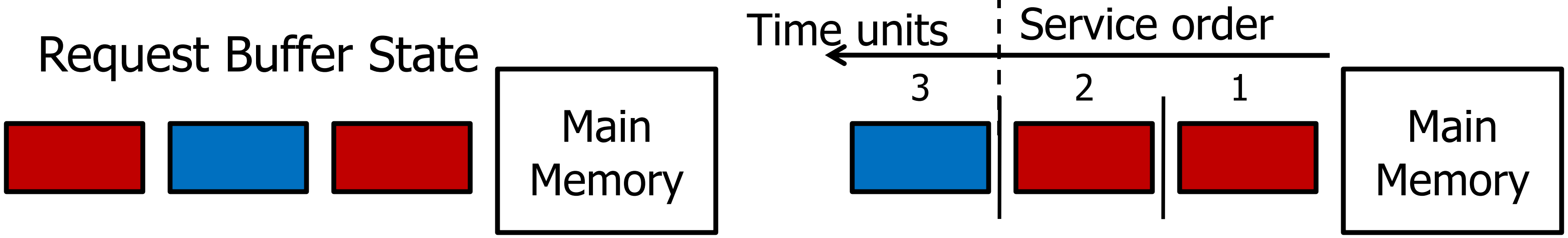
1. Run alone



2. Run with another application



3. Run with another application: highest priority

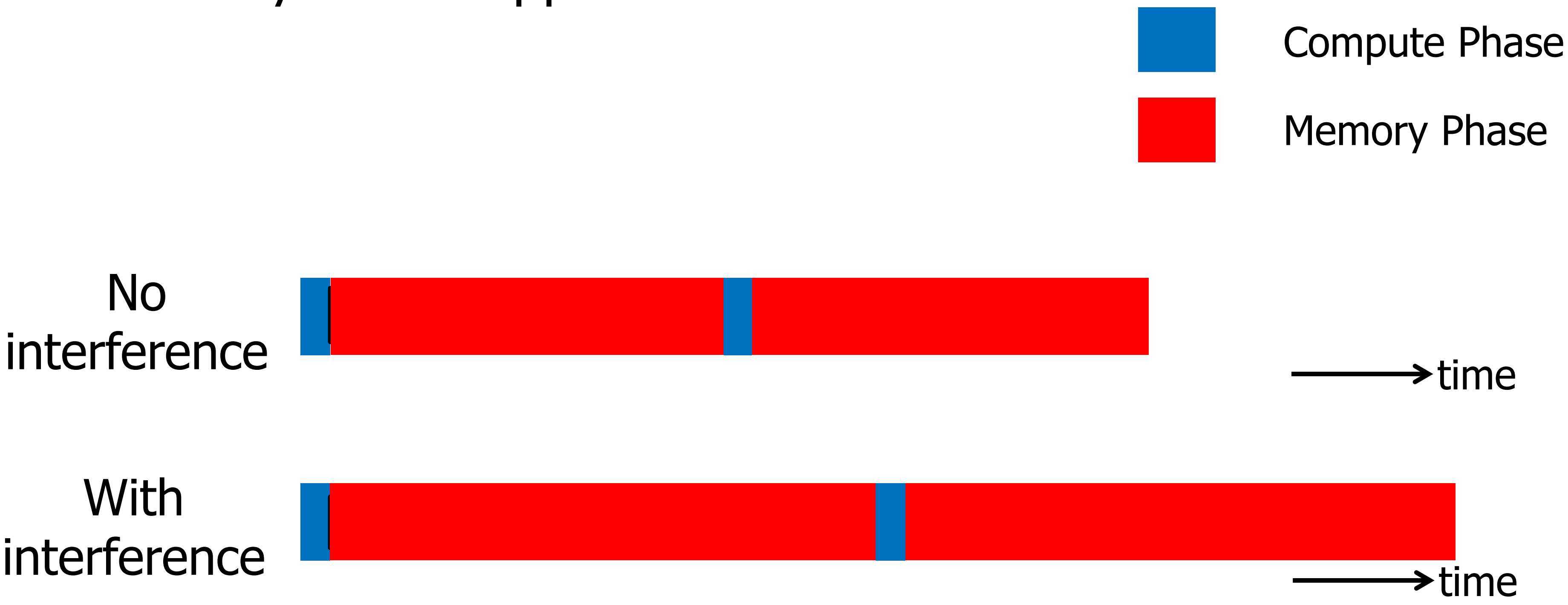


Memory Interference-induced Slowdown Estimation (MISE) model for memory bound applications

$$\text{Slowdown} = \frac{\text{Request Service Rate}_{\text{Alone}}\ (\text{RSR}_{\text{Alone}})}{\text{Request Service Rate}_{\text{Shared}}\ (\text{RSR}_{\text{Shared}})}$$

Key Observation 3

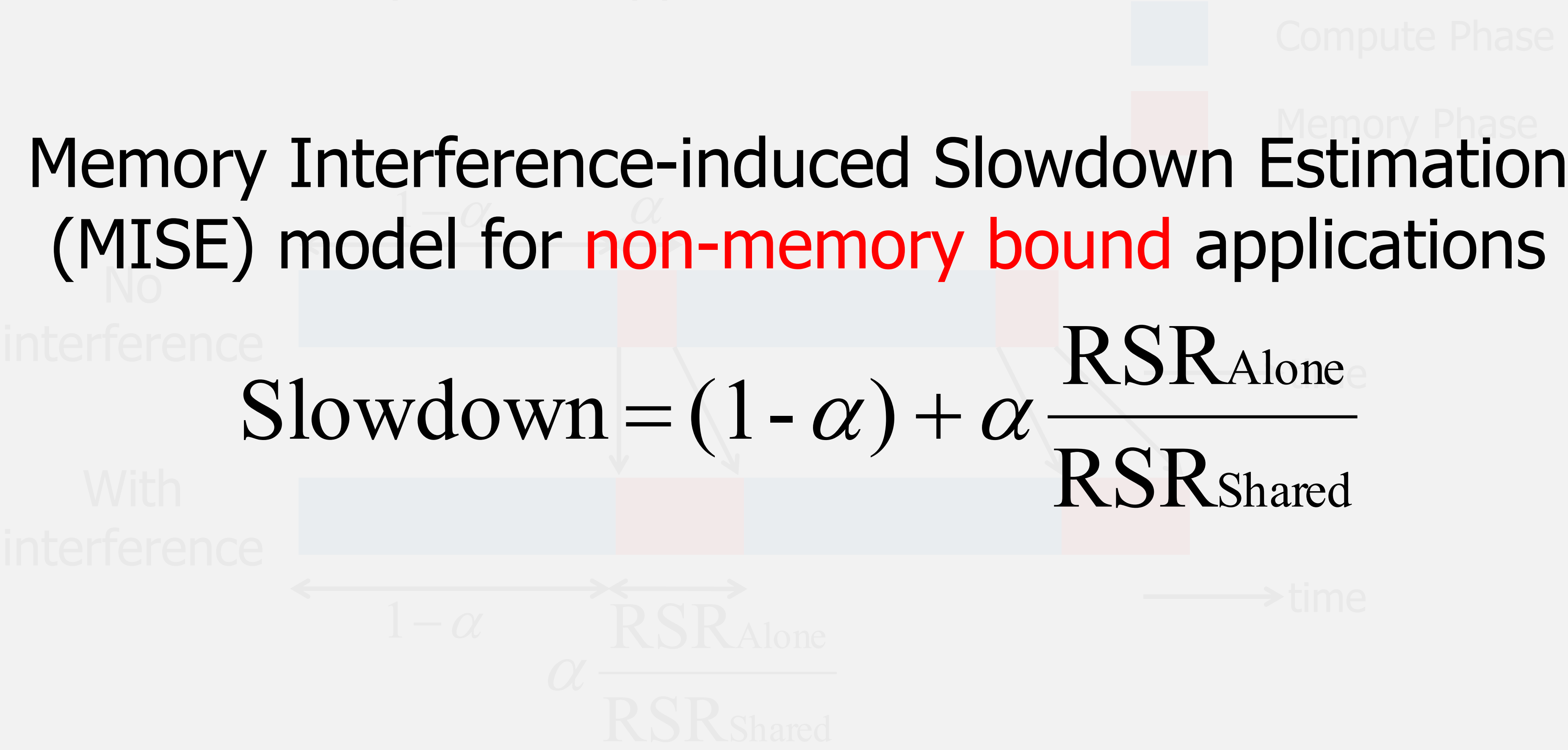
- Memory-bound application



Memory phase slowdown dominates overall slowdown

Key Observation 3

- Non-memory-bound application



Only memory fraction (α) slows down with interference

Memory Interference-induced Slowdown Estimation (MISE) model for non-memory bound applications

$$Slowdown = (1 - \alpha) + \alpha\frac{RSR_{Alone}}{RSR_{Shared}}$$

Outline

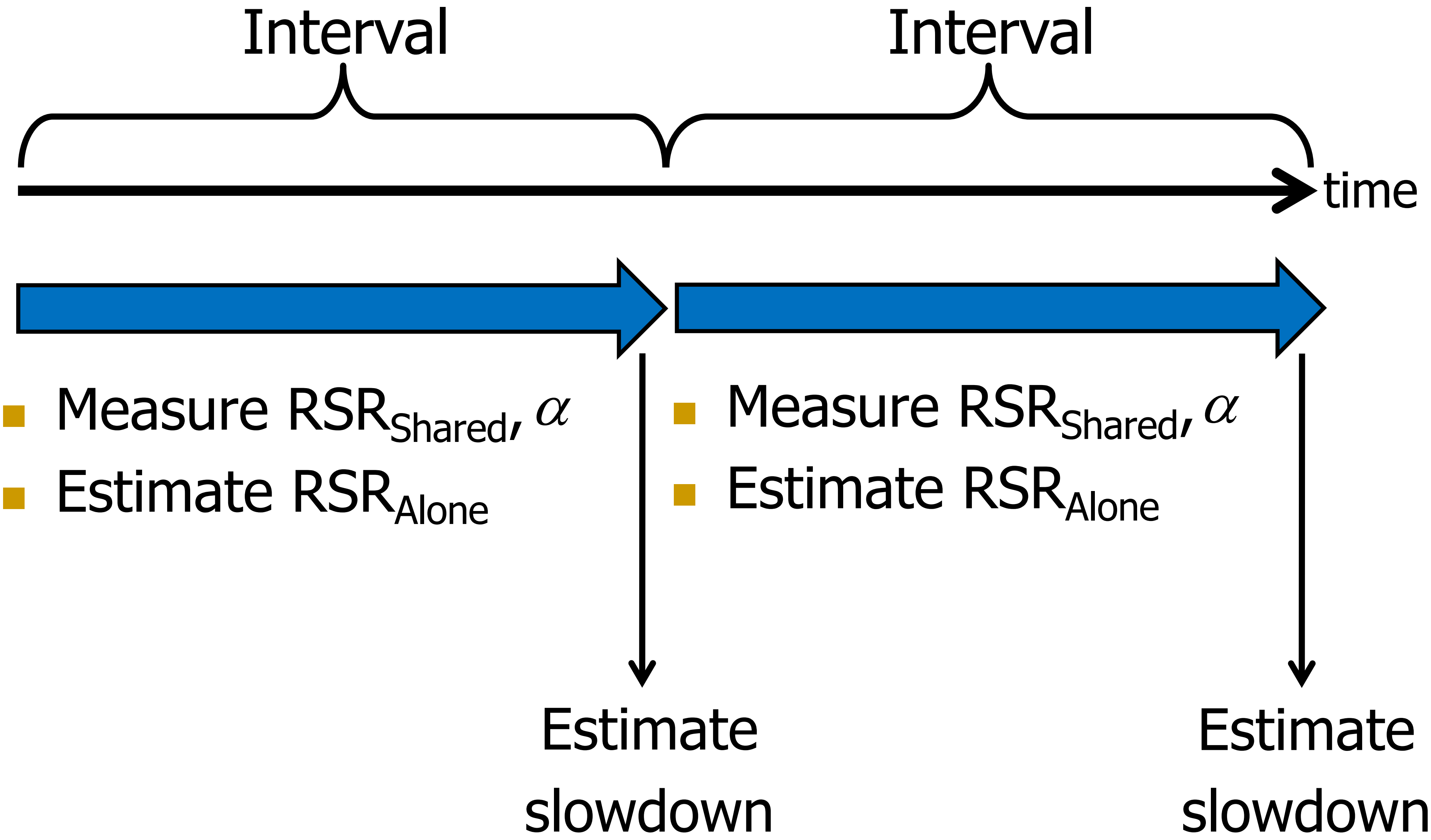
1. Estimate Slowdown

- Key Observations
- Implementation
- MISE Model: Putting it All Together
- Evaluating the Model

2. Control Slowdown

- Providing Soft Slowdown Guarantees
- Minimizing Maximum Slowdown

Interval Based Operation



Interval

Interval

time

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

Estimate slowdown

Estimate slowdown

Measuring $\mathrm{RSR}_{\mathrm{Shared}}$ and α

- Request Service Rate $_{\mathrm{Shared}}$ ($\mathrm{RSR}_{\mathrm{Shared}}$)
  - Per-core counter to track number of requests serviced
  - At the end of each interval, measure

$$\mathrm{RSR}_{\mathrm{Shared}} = \frac{\text{Number of Requests Serviced}}{\text{Interval Length}}$$

- Memory Phase Fraction (α)
  - Count number of stall cycles at the core
  - Compute fraction of cycles stalled for memory

Estimating Request Service Rate $_{\text{Alone}}$ (RSR$_{\text{Alone}}$)

- Divide each interval into shorter epochs

- At the beginning of each epoch
  - Memory controller randomly picks an application as the highest priority application

Goal: Estimate RSR$_{\text{Alone}}$

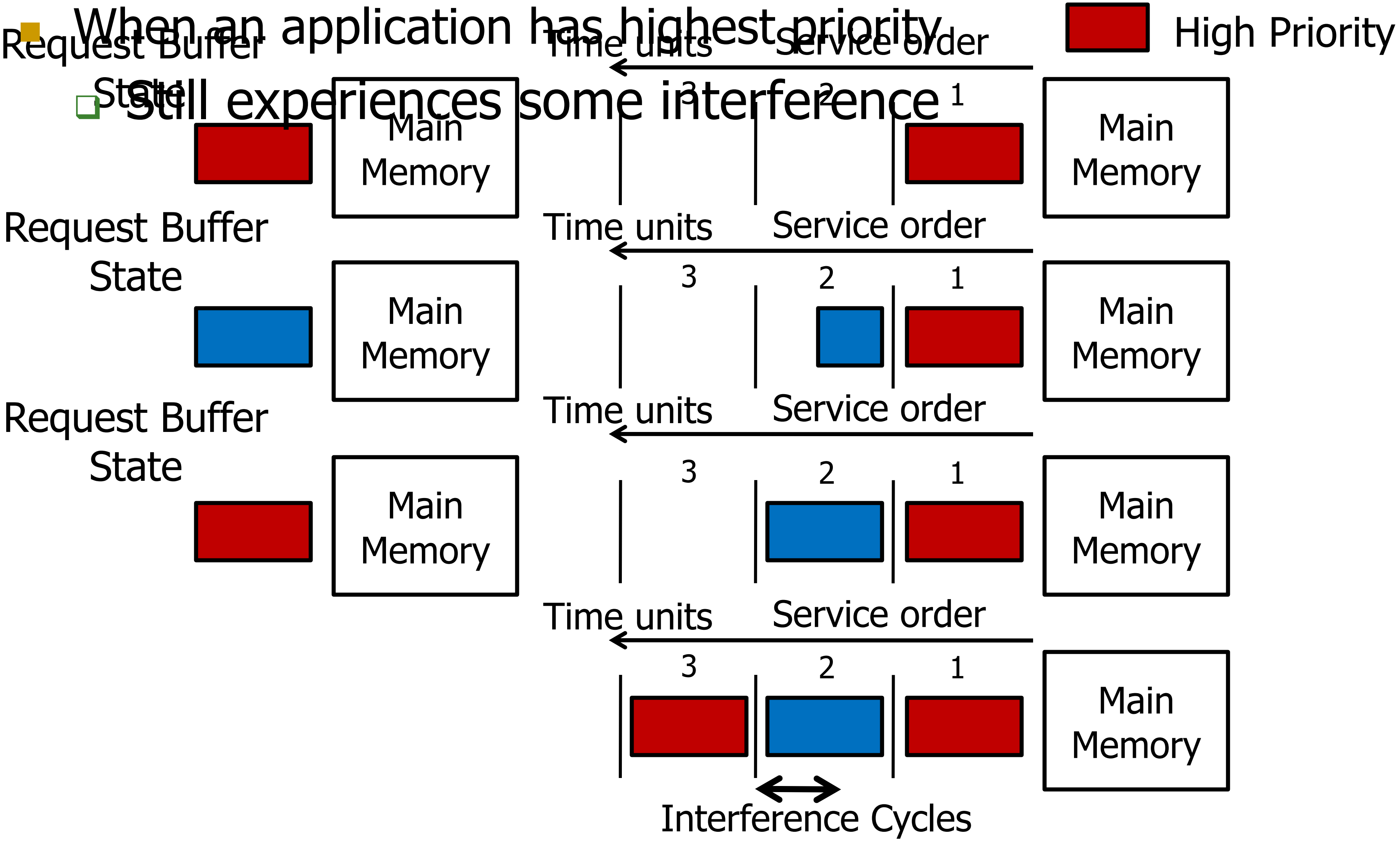
How: Periodically give each application highest priority in accessing memory

- At the end of an interval, for each application, estimate

$$\text{RSR}_{\text{Alone}} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority}}$$

Inaccuracy in Estimating RSR_{Alone}

- When an application has highest priority
  - Still experiences some interference



Accounting for Interference in RSR_{Alone} Estimation

- Solution: Determine and remove interference cycles from RSR_{Alone} calculation

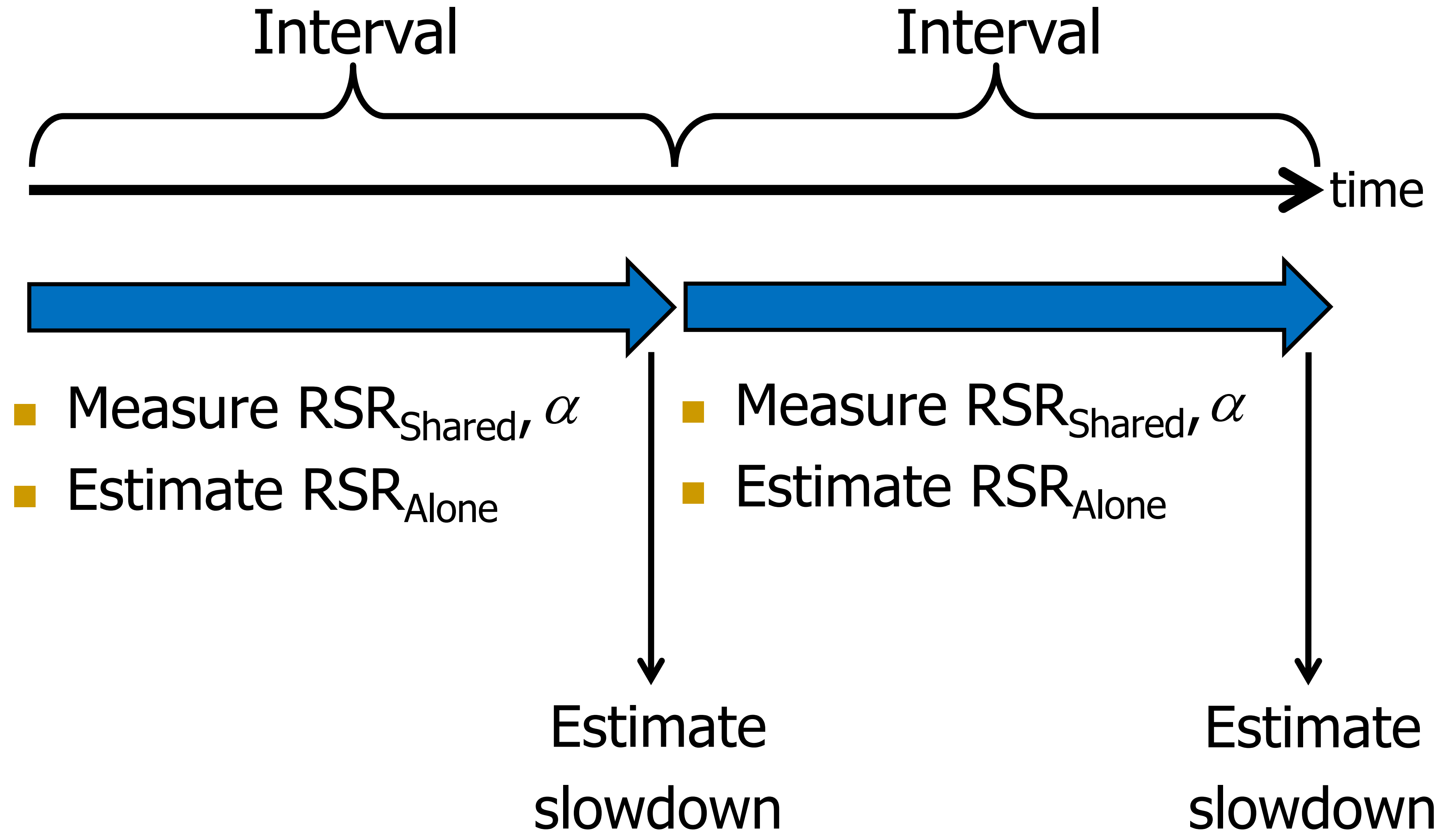
$$RSR_{Alone} = \frac{\text{Number of Requests During High Priority Epochs}}{\text{Number of Cycles Application Given High Priority} - \text{Interference Cycles}}$$

- A cycle is an interference cycle if
  - a request from the highest priority application is waiting in the request buffer *and*
  - another application's request was issued previously

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

MISE Model: Putting it All Together



Interval

Interval

time

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

- Measure RSR_{Shared}, α
- Estimate RSR_{Alone}

Estimate slowdown

Estimate slowdown

Outline

1. Estimate Slowdown
    - Key Observations
    - Implementation
    - MISE Model: Putting it All Together
    - Evaluating the Model
2. Control Slowdown
    - Providing Soft Slowdown Guarantees
    - Minimizing Maximum Slowdown

SAFARI

Previous Work on Slowdown Estimation

- Previous work on slowdown estimation
  - **STFM** (Stall Time Fair Memory) Scheduling [Mutlu+, MICRO '07]
  - **FST** (Fairness via Source Throttling) [Ebrahimi+, ASPLOS '10]
  - **Per-thread Cycle Accounting** [Du Bois+, HiPEAC '13]

- Basic Idea:

$$\text{Slowdown} = \frac{\boxed{\text{Stall Time}_{\text{Alone}}} \nearrow \text{Hard}}{\text{Stall Time}_{\text{Shared}} \searrow \text{Easy}}$$

Count number of cycles application receives interference

Two Major Advantages of MISE Over STFM

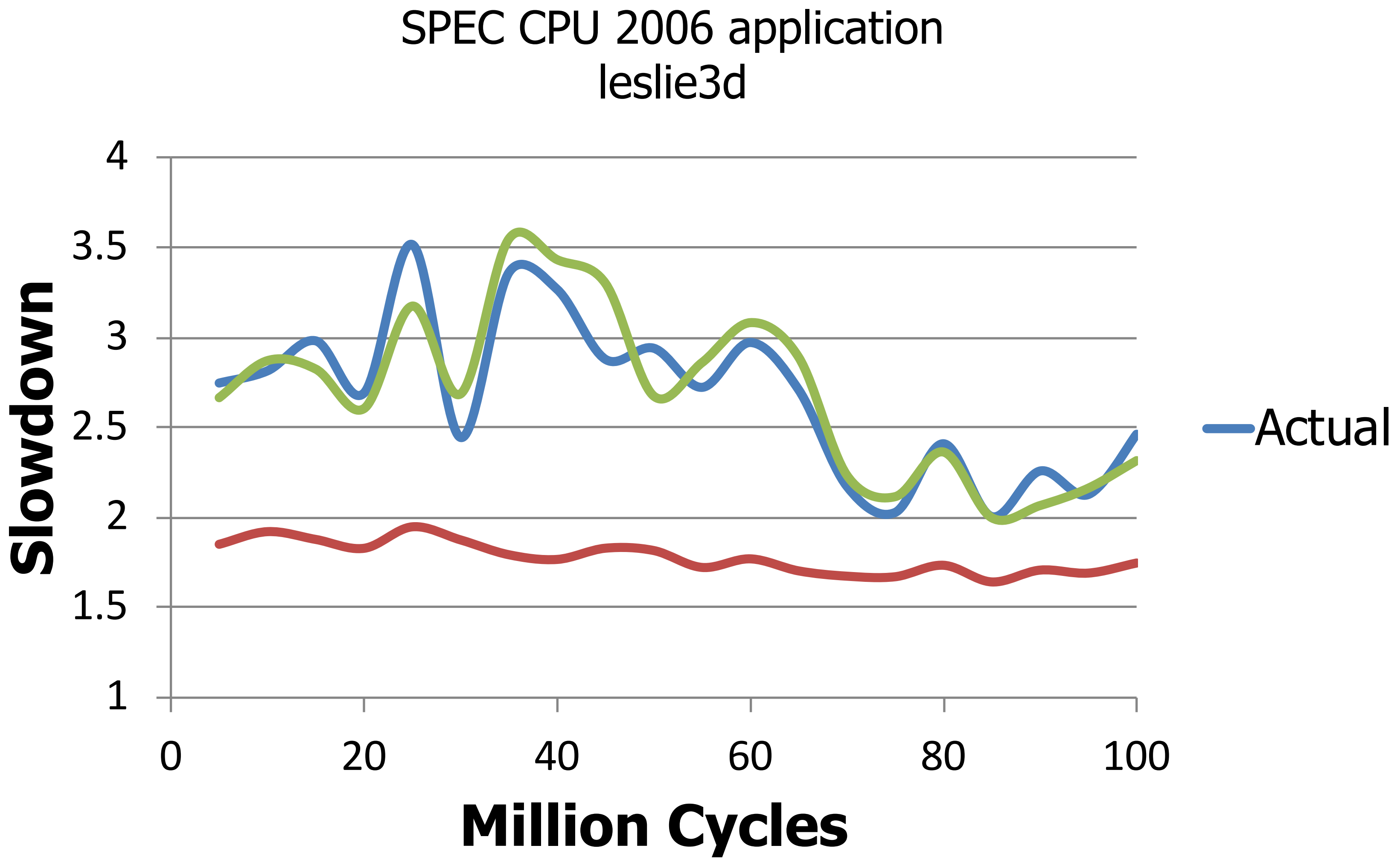
- Advantage 1:
  - STFM estimates alone performance while an application is receiving interference → Hard
  - MISE estimates alone performance while giving an application the highest priority → Easier
- Advantage 2:
  - STFM does not take into account compute phase for non-memory-bound applications
  - MISE accounts for compute phase → Better accuracy

Methodology

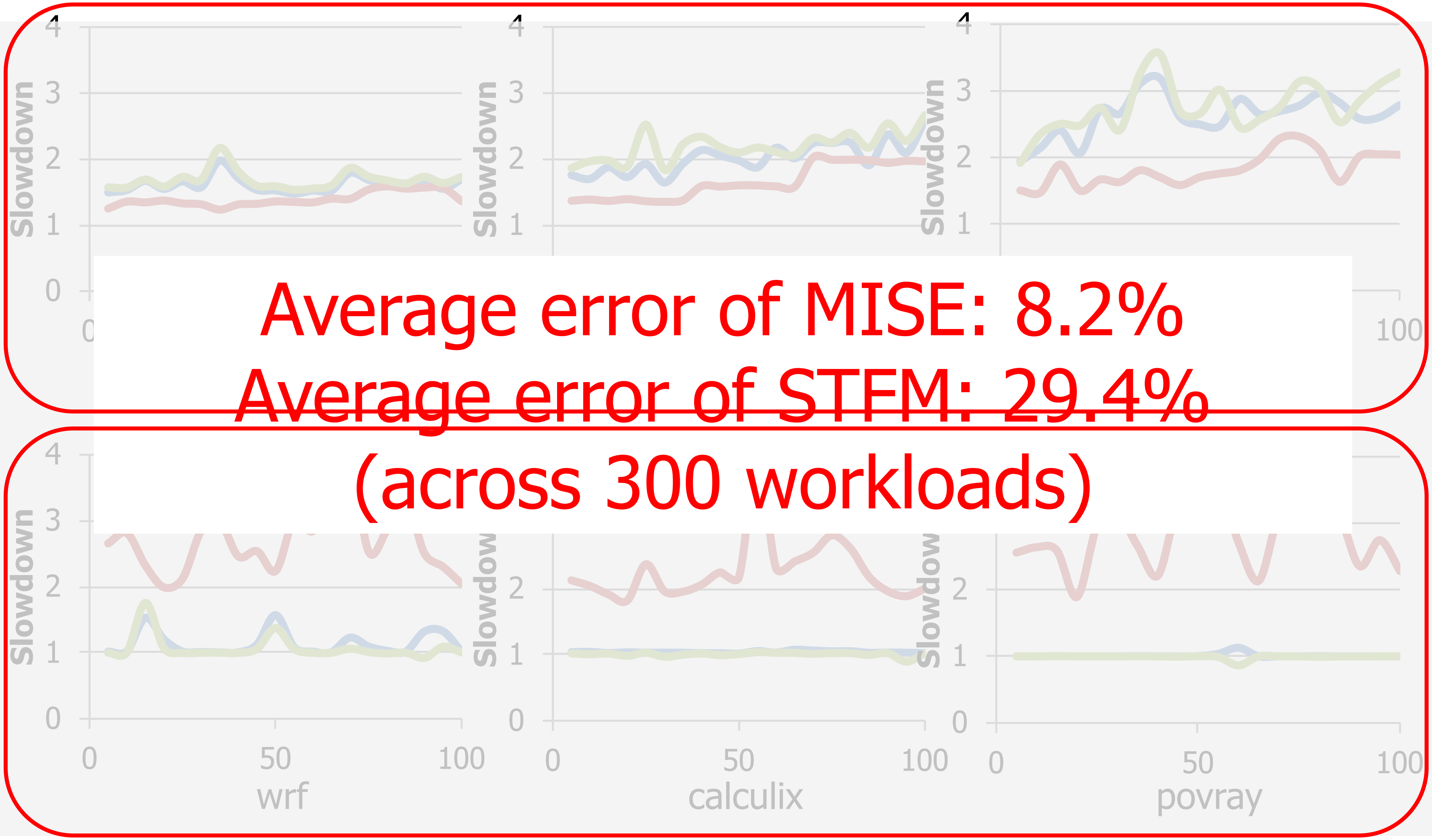
- Configuration of our simulated system
  - 4 cores
  - 1 channel, 8 banks/channel
  - DDR3 1066 DRAM
  - 512 KB private cache/core

- Workloads
  - SPEC CPU2006
  - 300 multi programmed workloads

Quantitative Comparison



Comparison to STFM



Average error of MISE: 8.2%
Average error of STFM: 29.4%
(across 300 workloads)

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Providing "Soft" Slowdown Guarantees

- Goal

  1. Ensure QoS-critical applications meet a prescribed slowdown bound
  2. Maximize system performance for other applications

- Basic Idea
  - Allocate just enough bandwidth to QoS-critical application
  - Assign remaining bandwidth to other applications

MISE-QoS: Mechanism to Provide Soft QoS

- Assign an initial bandwidth allocation to QoS-critical application
- Estimate slowdown of QoS-critical application using the MISE model
- After every N intervals
  - If slowdown > bound B +/- ε, increase bandwidth allocation
  - If slowdown < bound B +/- ε, decrease bandwidth allocatior
- When slowdown bound not met for N intervals
  - Notify the OS so it can migrate/de-schedule jobs

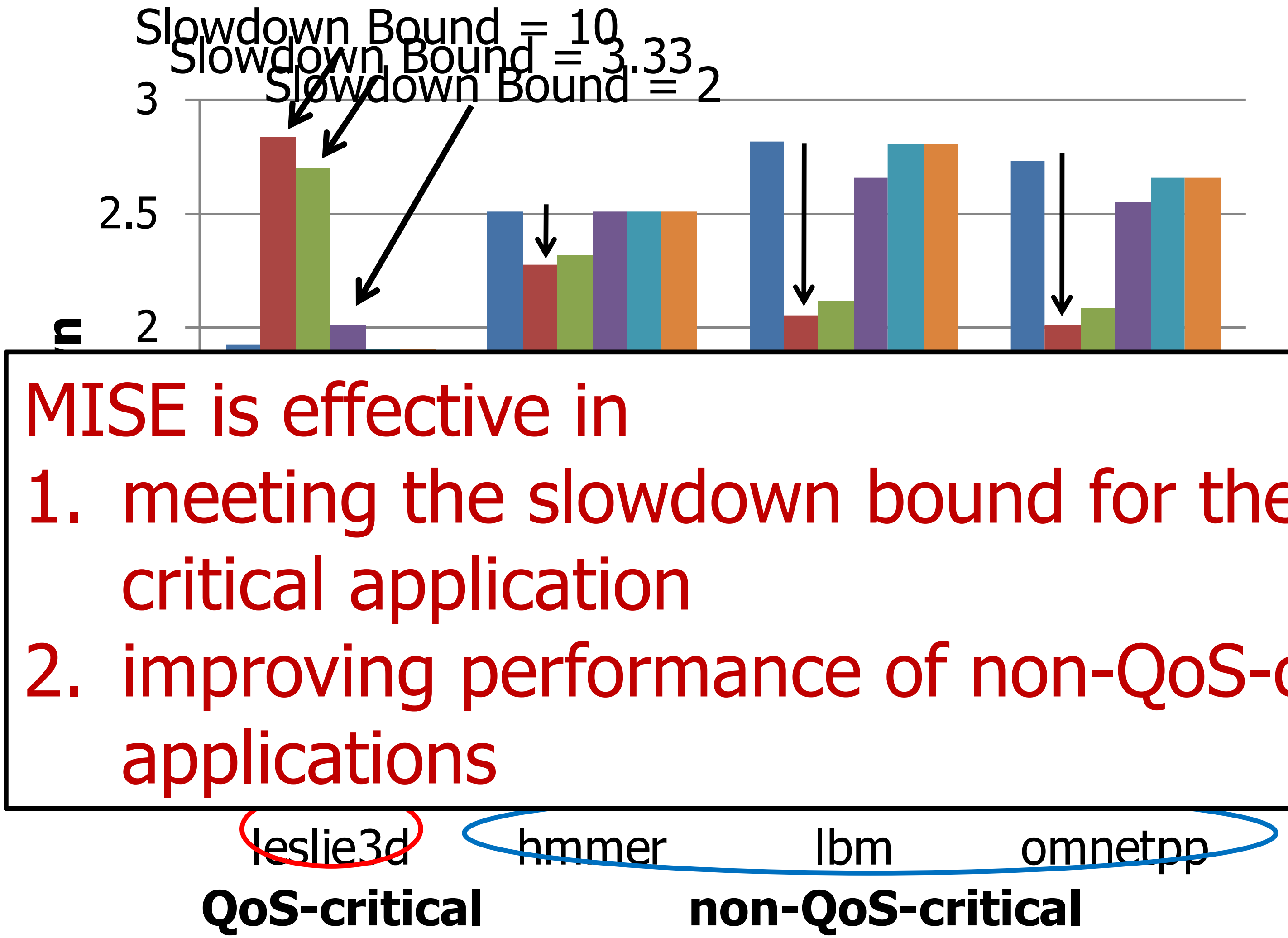
Methodology

- Each application (25 applications in total) considered the QoS-critical application
- Run with 12 sets of co-runners of different memory intensities
- Total of 300 multiprogrammed workloads
- Each workload run with 10 slowdown bound values
- Baseline memory scheduling mechanism
  - Always prioritize QoS-critical application

    [Iyer+, SIGMETRICS 2007]
  - Other applications' requests scheduled in FRFCFS order

    [Zuravleff +, US Patent 1997, Rixner+, ISCA 2000]

A Look at One Workload



MISE is effective in

1. meeting the slowdown bound for the QoS-critical application
2. improving performance of non-QoS-critical applications

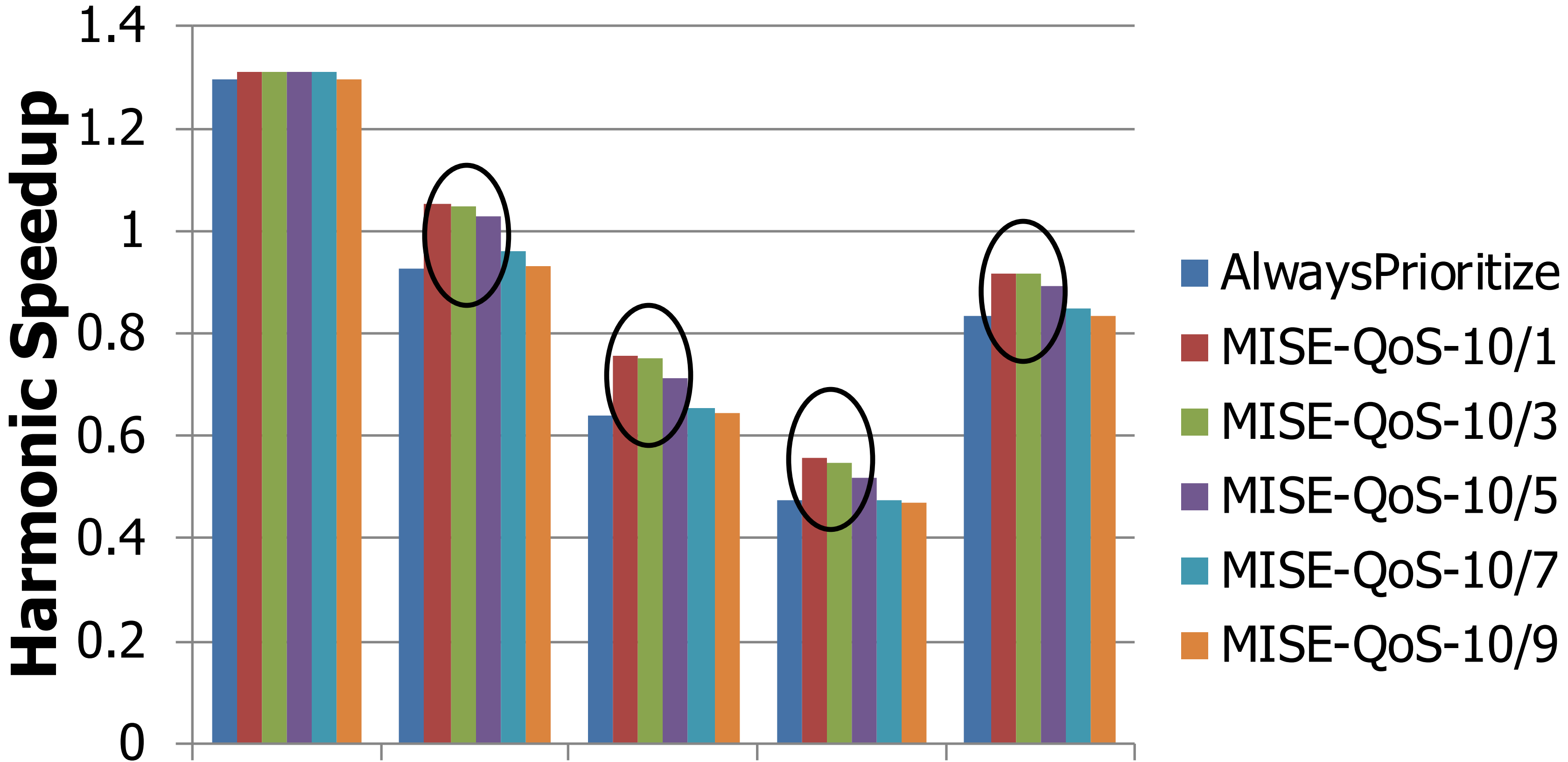
Effectiveness of MISE in Enforcing QoS

Across 3000 data points

	Predicted Met	Predicted Not Met
QoS Bound Met	78.8%	2.1%
QoS Bound Not Met	2.2%	16.9%

MISE-QoS correctly predicts whether or not the bound is met for 95.7% of workloads

Performance of Non-QoS-Critical Applications



When slowdown bound is 10/3
MISE-QoS improves system performance by 10%

Outline

1. Estimate Slowdown
   - Key Observations
   - Implementation
   - MISE Model: Putting it All Together
   - Evaluating the Model
2. Control Slowdown
   - Providing Soft Slowdown Guarantees
   - Minimizing Maximum Slowdown

Other Results in the Paper

- Sensitivity to model parameters
  - Robust across different values of model parameters

- Comparison of STFM and MISE models in enforcing soft slowdown guarantees
  - MISE significantly more effective in enforcing guarantees

- Minimizing maximum slowdown
  - MISE improves fairness across several system configurations

Summary

- Uncontrolled memory interference slows down applications unpredictably
- Goal: Estimate and control slowdowns
- Key contribution
  - MISE: An accurate slowdown estimation model
  - Average error of MISE: 8.2%
- Key Idea
  - Request Service Rate is a proxy for performance
  - Request Service Rate $_{\text{Alone}}$ estimated by giving an application highest priority in accessing memory
- Leverage slowdown estimates to control slowdowns
  - Providing soft slowdown guarantees
  - Minimizing maximum slowdown

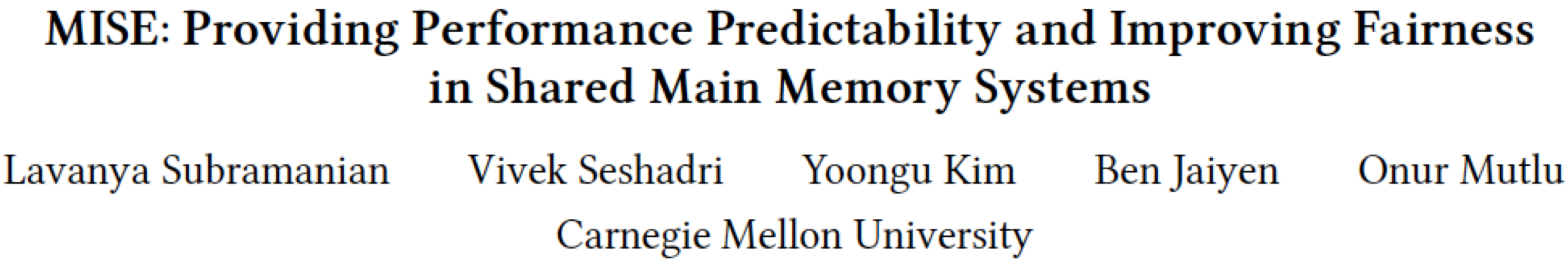
MISE: Pros and Cons

- Upsides:
  - Simple new insight to estimate slowdown
  - Much more accurate slowdown estimations than prior techniques (STFM, FST)
  - Enables a number of QoS mechanisms that can use slowdown estimates to satisfy performance requirements

- Downsides:
  - Slowdown estimation is not perfect - there are still errors
  - Does not take into account caches and other shared resources in slowdown estimation

More on MISE

- Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu, **"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"** *Proceedings of the 19th International Symposium on High-Performance Computer Architecture* (***HPCA***), Shenzhen, China, February 2013. Slides (pptx)



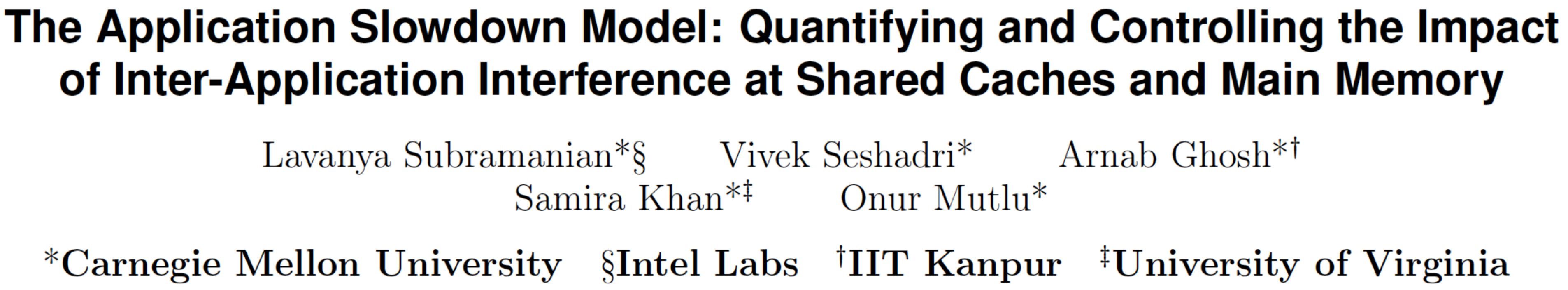
MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems

Lavanya Subramanian Vivek Seshadri Yoongu Kim Ben Jaiyen Onur Mutlu

Carnegie Mellon University

Extending MISE to Shared Caches: ASM

- Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu,
  "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory"
  Proceedings of the 48th International Symposium on Microarchitecture (***MICRO***), Waikiki, Hawaii, USA, December 2015.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pptx) (pdf)] [Poster (pptx) (pdf)]
  [Source Code]



The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory

Lavanya Subramanian*§ Vivek Seshadri* Arnab Ghosh*†
Samira Khan*‡ Onur Mutlu*

*Carnegie Mellon University §Intel Labs †IIT Kanpur ‡University of Virginia

Handling Memory Interference In Multithreaded Applications

Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin,
Chang Joo Lee, Onur Mutlu, and Yale N. Patt,
"Parallel Application Memory Scheduling"
Proceedings of the 44th International Symposium on Microarchitecture (***MICRO***),
Porto Alegre, Brazil, December 2011. Slides (pptx)

Multithreaded (Parallel) Applications

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications

- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, …

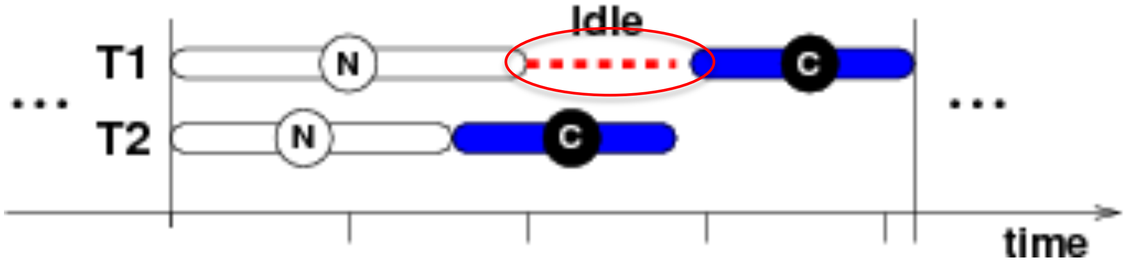
- Some threads can be on the critical path of execution due to synchronization; some threads are not

- Even within a thread, some “code segments” may be on the critical path of execution; some are not

Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

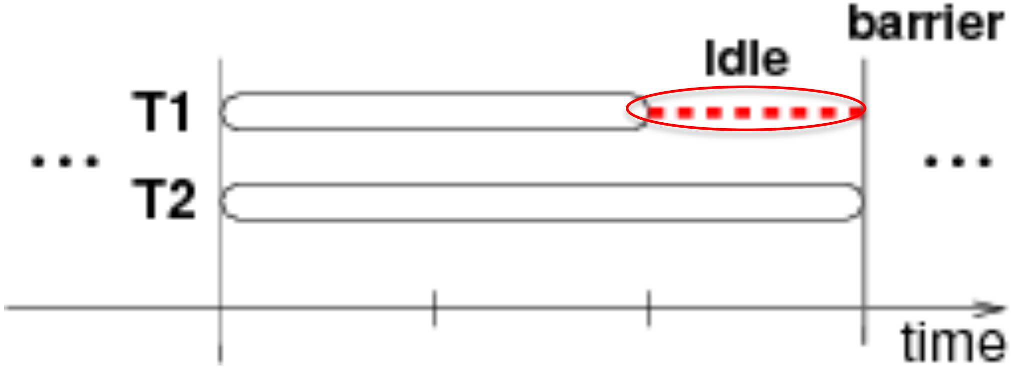
```
Each thread:
  loop {
    Compute                   N
    lock(A)
      Update shared data
    unlock(A)                 C
  }
```



Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving at the barrier is on the critical path

```
Each thread:
    loop1 {
        Compute
    }
    barrier
    loop2 {
        Compute
    }
```



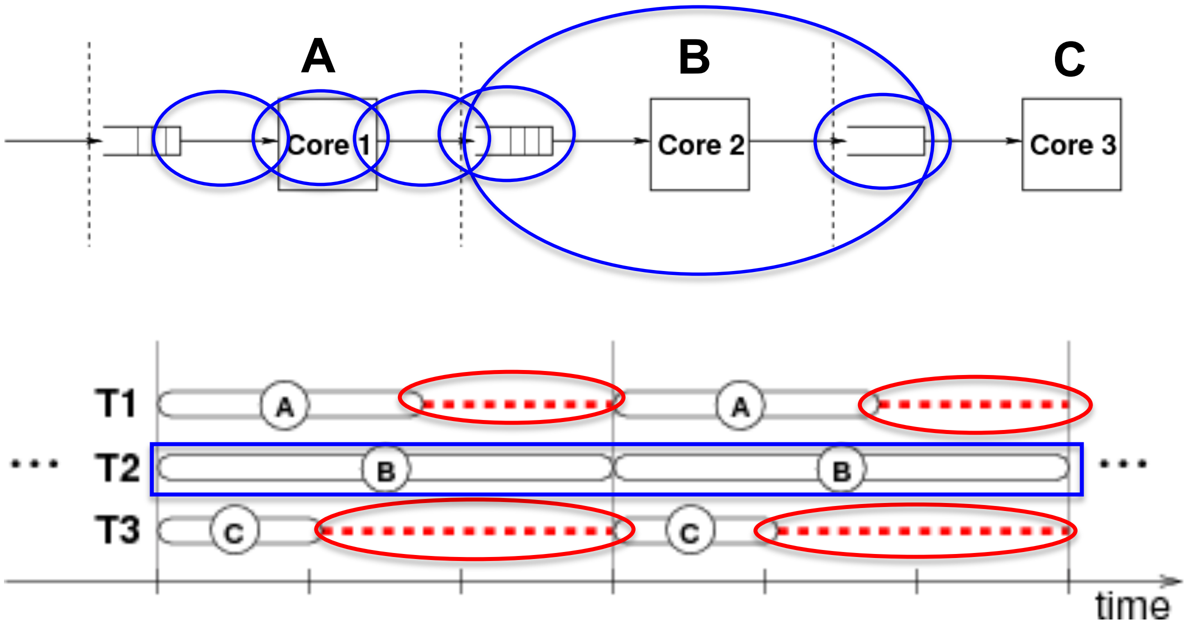
Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*
- Threads execute stages on different cores
- Thread executing the slowest stage is on the critical path

```
loop {
  Compute1     A

  Compute2     B

  Compute3     C
}
```



Handling Interference in Parallel Applications

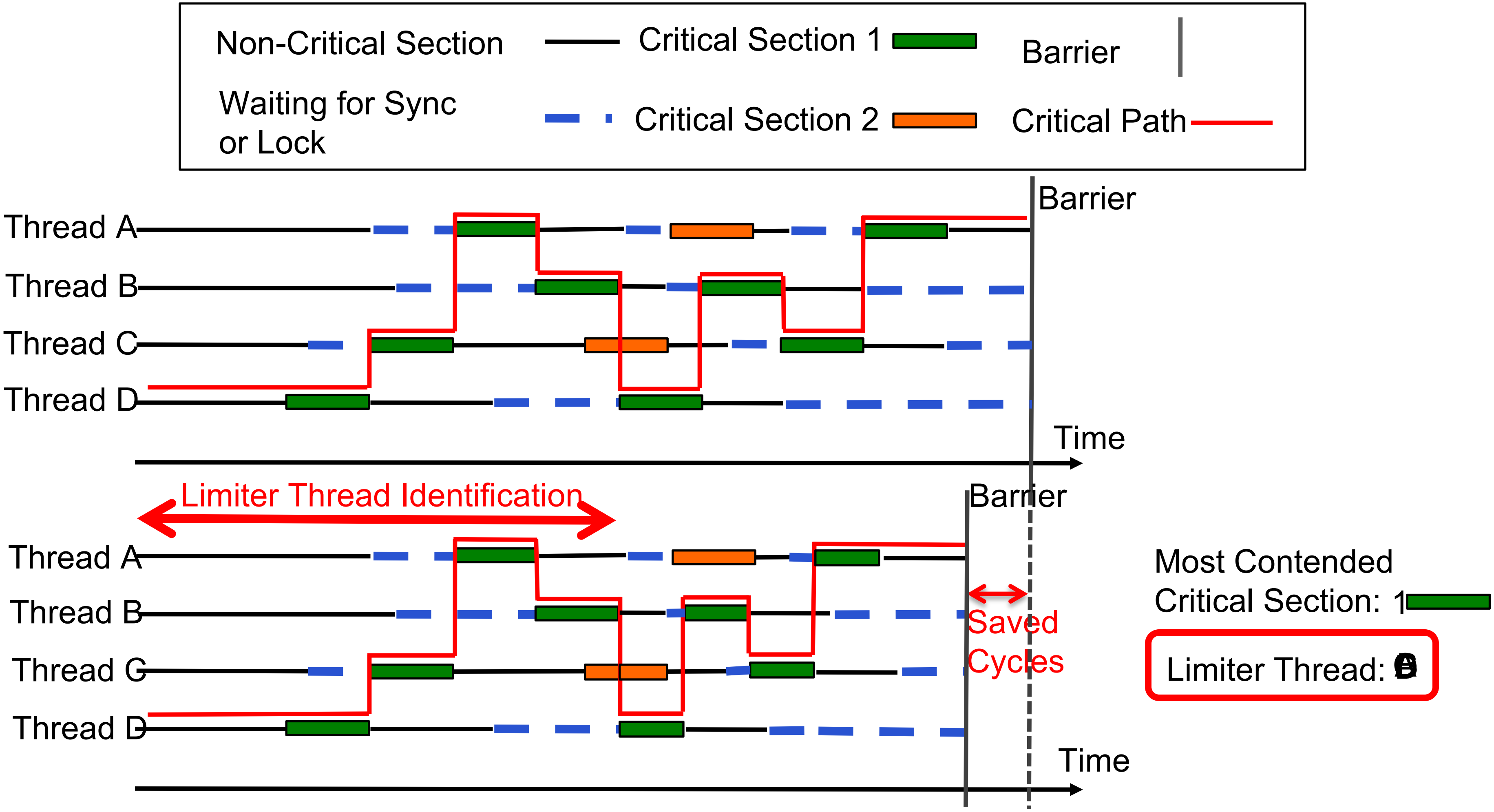
- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO'11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread executing the slowest pipeline stage
  - Thread that is falling behind the most in reaching a barrier

PAMS Micro 2011 Talk

Prioritizing Requests from Limiter Threads

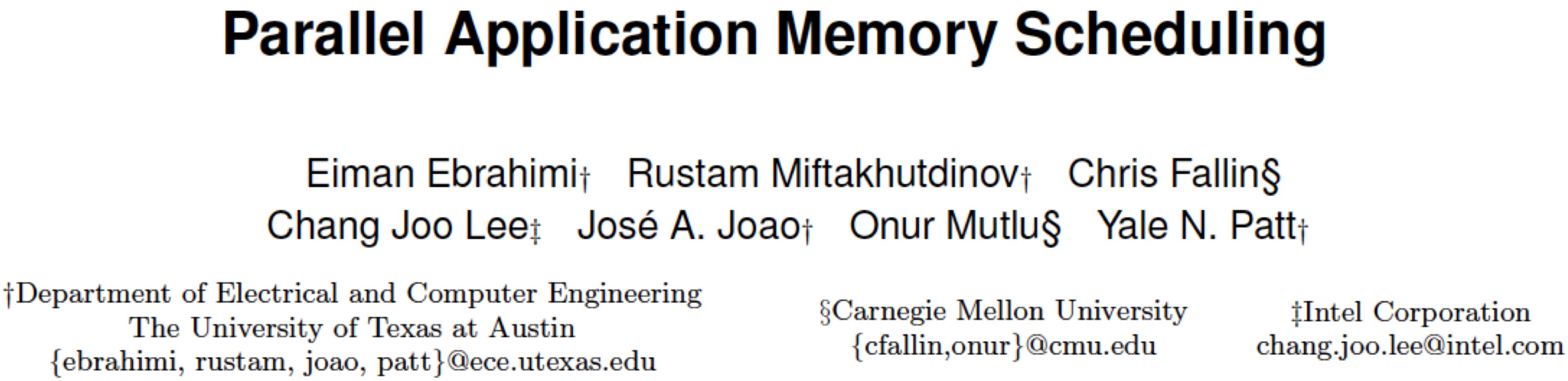


Parallel App Mem Scheduling: Pros and Cons

- Upsides:
  - Improves the performance of multi-threaded applications
  - Provides a mechanism for estimating “limiter threads”
  - Opens a path for slowdown estimation for multi-threaded applications
- Downsides:
  - What if there are multiple multi-threaded applications running together?
  - Limiter thread estimation can become complex

More on PAMS

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt, **"Parallel Application Memory Scheduling"** *Proceedings of the 44th International Symposium on Microarchitecture (**MICRO**)*, Porto Alegre, Brazil, December 2011. Slides (pptx)



Parallel Application Memory Scheduling

Eiman Ebrahimi[†] Rustam Miftakhutdinov[†] Chris Fallin[§]
Chang Joo Lee[‡] José A. Joao[†] Onur Mutlu[§] Yale N. Patt[†]

[†]Department of Electrical and Computer Engineering
The University of Texas at Austin
{ebrahimi, rustam, joao, patt}@ece.utexas.edu

[§]Carnegie Mellon University
{cfallin,onur}@cmu.edu

[‡]Intel Corporation
chang.joo.lee@intel.com